MK++ Kernel Interfaces

OSF

November 17, 1995

Revision History:

Revision 1	November 19, 1993	
Revision 1.1	May 26, 1994	Consistency with high level design
Revision 1.2	September 15, 1994	Consistency with code
Revision 1.3	May 5, 1995	Consistency with first release
Revision 1.4	November 17, 1995	Formal evaluation version

This document is partially derived from earlier Mach documents written by Robert V. Baron, Joseph S. Barrera, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, Alessandro Forin, David B. Golub, Richard F. Rashid, Mary R. Thompson, Avadis Tevanian, Jr., Michael W. Young.

This work was sponsored in part by the Advanced Research Projects Agency and the Air Force Materiel Command.

Contents

CHAPTER 1 Introduction

This book documents the various interfaces to the MK++ kernel. The text describes each interface to the kernel in isolation. The relationship of interfaces to one another, and the way that interfaces are combined to write user servers is the subject of companion volumes.

The organization of this book is such that it follows the organization of the kernel into its major functional areas. Although the kernel interface is itself not object oriented, the division of interfaces into areas is largely done according to the significant object utilized or manipulated by the interfaces. Each such object receives its own chapter. Of course, the assignment of interfaces into these chapters is a subjective process. An interface that requires rights for two ports of two different types could be grouped with the set of interfaces associated with either object type. Each interface, though, appears only once in this book.

Appendices give a description of the structures and fields used by these interfaces, a list of possible error return values from the kernel, an alphabetical index by object type and one by function and data structure name.

Interface Descriptions

Each interface is listed separately, each starting on its own page. For each interface, some or all of the following features are presented:

- The name of the interface
- A brief description
- A synopsis of the interface, in C form
- Any macro or special forms of the call

- An extended description of the function performed by the call
- A description of each parameter to the call
- Additional notes on the use of the interface
- Cautions relating to the interface use
- An explanation of the significant return values
- References to related interfaces or data structures

Interface Types

Most of the interfaces in this book are MIG generated interfaces. That is, they are stub routines generated from MIG interface description files. Calling these interfaces will actually result in a Mach IPC message being sent to the port that is the first argument in the call. This has two important effects.

- These calls may fail for various MIG or IPC related reasons. The list of error returns for these calls should always be considered to also include the IPC related errors (MACH_MSG_..., MACH_SEND_... and MACH_RCV_...) and the MIG related errors (MIG_...).
- These calls only produce their expected effect when the acting port is indeed a port of the specified type. That is, if a call expects a port that names a task (a kernel task port) and the port is instead a port managed by a task, the MIG stub routine will still happily generate the appropriate Mach message and send it to that task. What the target task will do with the message is up to it. Note that it is this effect that allows servers to redirect messages transparently.

A few of these interfaces are actually system calls (traps). In general, the system calls (with the obvious exception of the **mach_msg** call) work only on the current task or thread. Any routine not documented as a system call is a MIG stub routine.

Most of these interfaces are of the type **Function**. This means that there is actually a C callable function (most likely in **libmach_sa.a**) that has the calling sequence listed and that when called invokes some kernel or kernel related service. If the interface is a system trap instead of a message, it will be listed as a **System Trap**.

Some interfaces have the type **Server Interface**. Such a description applies to interfaces that are called in server tasks on behalf of messages sent from the kernel. That is, it is assumed that some task is listening on a port to which the kernel is to send messages. A received message will be passed to a MIG generated server routine (*service*_**server**) that will call an appropriate server target function. It is these server target functions, one for each different message that the kernel generates, that are listed as **Server Interface**s. For any given kernel message, there are any number of possible server interface calling sequences that can be generated, by permuting the order of the data provided in the message, omitting some data elements or including or omitting various message field elements (such as sequence numbers). In most cases, a single server interface calling sequence has been chosen with a given MIG generated server message de-multiplexing routine that calls these interfaces. In some cases, there are more than one MIG generated server routines that call upon different server interfaces associated with that MIG service routine. In any event, all **Server Interfaces** contain within their documentation the name of the MIG generated server routine that invokes the interface.

Special Forms

There are various special interface forms defined in this volume.

- The **Macro** form specifies macros (typically defined in **mach.h**) that provide shorthand equivalents for some variations of the longer function call.
- The **Sequence Number** form of a **Server Interface** defines an additional MIG generated interface that supplies the sequence number from the message causing the server interface to be invoked. The existence of such a form implies the existence of an alternate MIG generated message de-multiplexing routine that invokes this special interface form.
- The **Asynchronous** form defines a MIG generated version of a **Function** that allows the function to be invoked asynchronously. Such a version requires an additional parameter to specify the reply port to which the reply is sent. The return value from the asynchronous function is the return status from the **mach_msg** call sending the request, not the resulting status of the kernel operation. The asynchronous interface also requires a matching **Server Interface** that defines the reply message containing data that would have been output values from the normal function, as well as the resulting status from the kernel operation.
- The **Expanded** form specifies a variant of an interface that takes additionally parameters uncommonly used and therefore not supplied as parameters to the base call.
- The **In-line** form specifies a variant of an interface that passes data in-line as opposed to out-of-line.

Parameter Types

Each interface description supplies the C type of the various parameters. The parameter descriptions then indicate whether these parameters are input ("in"), output ("out") or both ("in/out"). This information appears in square brackets before the parameter description. Additional information also appears within these brackets for special or non-obvious parameter conventions.

The most common notation is "scalar", which means that the parameter somehow derives from an *int* type.

Port types are scalar types but are marked specifically as to the type of port named by the parameter and the type of right (receive, send or send-once) required. A send right is virtually always copied (with **memory_object_representative** being the notable exception). A receive right is almost always converted to some other right (as indicated in parenthesis in the description). The "target" of a server interface is always listed as a "(receive) right" since the target right name will be that of the receive right.

If the notation says “structure”, the parameter is a direct structure type whose layout is described in APPENDIX C.

The notation “pointer to in array/structure/scalar” means that the caller supplies a pointer to the data. Arrays always have this property following from C language rules. If not so noted, input parameters are passed by value.

Output parameters are always passed by reference following C language rules. Hence the notation “out array/structure/scalar” actually means that the caller must supply a pointer to the storage to receive the output value. If a parameter is in/out, the notation “pointer to in/out array/structure/scalar” will appear. Since the parameter is also an output parameter, it must be passed by reference, hence it appears as a “pointer to in array/structure/scalar” when used as an input parameter.

In contrast, the notation “out pointer to dynamic array” means that the kernel will allocate space for returned data (as if by **vm_allocate**) and will modify the pointer named by the output parameter (that is, the parameter to the function is a pointer to a pointer) to point to this allocated memory. The task should **vm_deallocate** this space when done referencing it.

For a Server Interface, the corresponding version of the above is “in pointer to dynamic array”. This indicates that the kernel has allocated space for the data (as if by **vm_allocate**) and is supplying a pointer to the data as the input parameter to the server interface routine. It is the job of the server interface routine to arrange for this data to be **vm_deallocate**d when the data is no longer needed.

Error Return Values

APPENDIX D documents the various error return values defined by the kernel. However, since the kernel interfaces are actually MIG generated stubs that send IPC messages, the set of errors that is possible for any given interface is quite extensive although few possibilities are seen in practice.

The various functions described in this volume (with the exception of the system traps) are MIG generated stub subroutines. As such, if the number of parameters or their sizes is incorrect, the stub may fail in a machine dependent way as would any other subroutine.

Assuming the correct number and size of the parameters, the MIG stub will simply marshall these values, making no consistency checks. The stub then attempts to send this message using **mach_msg**. As such, the various IPC errors (MACH_SEND_...) are possible.

If the destination port is valid but names a port whose receive right is held by a task, the stub generated message will be sent to that task; what the task will do is up to it. Assuming that the destination port does name a kernel object, the message will go to the kernel. If the message is not one that object accepts, the caller will get KERN_INVALID_ARGUMENT. For operations that bind two objects (such as **memory_object_establish**) when an additional right is sent for the purposes of asserting privilege, or when the additional right itself is being manipulated, specific error return values are generated if the port is of the wrong type.

Invalid non-port parameter values return the error KERN_INVALID_VALUE if their value is inherently ill-formed or out of range, but return specific error values if the value is not permitted at this point in time (such as a port name that is a valid name, but does not currently name a valid right).

Each kernel subsystem defines its own interesting set of errors, which are listed for the relevant interfaces. Generic or messaging errors are not listed for each interface; only those specific to that interface's functioning are listed.

A return value of KERN_SUCCESS (or any other equivalent value) indicates that the requested (synchronous) operation was performed and any return values returned. In the case of asynchronous requests, KERN_SUCCESS indicates (on the request side) only that the IPC message was delivered, or (on the response side) that the operation was performed and any return values returned.

MIG Server Routines

For each logical set of messages sent from the kernel to server interfaces, there is one or more MIG-generated server routines used to parse the message and call the appropriate server routine.

Each of these routines take two parameters: a pointer to a request message and a pointer to a reply message.

boolean_t *group*_**server**
(mach_msg_header_t* *in_msg,*
mach_msg_header_t* *out_msg*);

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. The result of the server routine call indicates whether the message was a legitimate message that was accepted by the target routine.

The currently supplied routines are the following. Each server interface description names the corresponding MIG server routine.

- **alarm_reply_server** — Handles kernel generated clock alarms
- **device_reply_server** — Handles messages from a kernel device driver
- **exception_server** — Handles kernel messages for an exception handler
- **host_reply_server** — Handles kernel messages for **device_open**
- **memory_cache_server_server** — Handles kernel messages for user pagers
- **memory_object_server_server** — Handles kernel messages for the default memory manager
- **notify_server** — Handles kernel generated IPC notifications

- **sample_server** — Handles kernel generated PC sample messages

CHAPTER 2 IPC Interface

This chapter discusses the specifics of the kernel's inter-"process" communication (IPC) interfaces. (Companion documents explain that the kernel does not provide the traditional notion of a process; however we use the term "inter-process" in the interest of consistency with analogous features in other systems.) The interfaces discussed are only the interfaces directly involved in sending and receiving IPC messages.

mach_msg

System Trap / Function — Sends and/or receives a message

SYNOPSIS

mach_msg_return_t **mach_msg**

(mach_msg_header_t*	*msg,*
mach_msg_option_t	*option,*
mach_msg_size_t	*send_size,*
mach_msg_size_t	*rcv_size,*
mach_port_t	*rcv_name,*
mach_msg_header_t *	*rcv_msg*);

DESCRIPTION

The **mach_msg** system call sends and/or receives Mach messages. Mach messages contain data, which can include port rights and addresses of regions of separately allocated memory.

If the *option* argument contains MACH_SEND_MSG, the call sends a message. The *send_size* argument specifies the size of the message buffer (header and body) to send. The *msgh_remote_port* field of the message header specifies the destination of the message.

If the *option* argument contains MACH_RCV_MSG, it receives a message. The *rcv_size* argument specifies the size of a buffer that will receive the message; messages that are larger are not received. The *rcv_name* argument specifies the port or port set from which to receive.

If the *option* argument contains both MACH_SEND_MSG and MACH_RCV_MSG, then **mach_msg** does both send and receive operations (in that order). If the send operation encounters an error (any return code other than MACH_MSG_SUCCESS), then the call returns immediately without attempting the receive operation. Semantically the combined call is equivalent to separate send and receive calls, but it saves a system call and enables other internal optimizations.

If the *option* argument specifies neither MACH_SEND_MSG nor MACH_RCV_MSG, then **mach_msg** does nothing.

PARAMETERS

msg

[pointer to in structure containing random and reply rights] A message buffer to be sent by **mach_msg**. This must be naturally aligned.

option

[in scalar] Message options are bit values, combined with bitwise-or. One or both of MACH_SEND_MSG and MACH_RCV_MSG should be used. Other options act as modifiers.

send_size

[in scalar] When sending a message, specifies the size of the message buffer to be sent (the size of the header and body) in bytes. Otherwise zero should be supplied.

rcv_size

[in scalar] When receiving a message, specifies the maximum size of the message buffer in bytes. Otherwise zero should be supplied.

rcv_name

[in random right] When receiving a message, specifies the port or port set. Otherwise MACH_PORT_NULL should be supplied.

rcv_msg

[pointer to out structure returning random and reply rights] A message buffer into which **mach_msg** will receive a message. This must be naturally aligned. The buffer can be the same as specified by *msg*.

NOTES

The Mach kernel provides message-oriented, capability-based inter-process communication. The inter-process communication (IPC) primitives efficiently support many different styles of interaction, including remote procedure calls, object-oriented distributed programming, streaming of data, and sending very large amounts of data.

Major Concepts

The IPC primitives operate on three abstractions: messages, ports, and port sets. User tasks access all other kernel services and abstractions via the IPC primitives.

The message primitives let tasks send and receive messages. Tasks send messages to ports. Messages sent to a port are delivered reliably (messages may not be lost) and are received in the order in which they were sent via send rights by a given sending task (or a given kernel). (Messages sent to send-once rights are unordered.) Messages contain a fixed-size header and a variable-sized message body containing kernel and user data. The header describes the destination and the size of the message (header plus body). The message body contains descriptions of additional port rights to be transmitted, descriptions of “out-of-line” memory regions to be sent and a variable amount of user data, which typically includes type conversion information. The out-of-line memory regions are (typically) disjoint from the message body.

The message can contain the addresses of regions of the sender's address space that should be transferred as part of the message. When a task receives a message containing such out-of-line regions of data, the data appears in unused portions of the receiver's address space. Under favorable circumstances, this transmission of out-of-line data is optimized so that the pages of data are transferred from sender to receiver without actually copying the data.

Ports hold a queue of messages. Tasks operate on a port to send and receive messages by exercising capabilities (rights) for the port. Multiple tasks can hold send rights for a port. Tasks can also hold send-once rights, which grant the ability to send a single message. Only one task can hold the receive capability (receive right) for a port. Port rights can be transferred between tasks via messages. The sender of a message can specify in the message that the message contains a port right. If a message contains a receive right for a port, then the receive right is removed from the sender of the message and the right is transferred to the receiver of the message. While the receive right is in transit, tasks holding send rights can still send messages to the port, and they are queued until a task acquires the receive right and uses it to receive the messages.

Tasks can receive messages from ports and port sets. The port set abstraction allows a single thread to wait for a message from any of several ports. Tasks manipulate port sets with a port set name, which is taken from the same name space as are the port rights. The port-set name may not be transferred in a message. A port set holds receive rights, and a receive operation on a port set blocks waiting for a message sent to any of the constituent ports. A port may not belong to more than one port set, and if a port is a member of a port set, the holder of the receive right can't receive directly from the port.

Port rights are a secure, location-independent way of naming ports. The port queue is a protected data structure, only accessible via the kernel's exported message primitives. Rights are also protected by the kernel; there is no way for a malicious user task to guess a port's internal name and send a message to a port to which it shouldn't have access. Port rights do not carry any location information. When a receive right for a port moves from task to task, and even between tasks on different machines, the send rights for the port remain unchanged and continue to function.

Port Rights

Each task has its own space of port rights. Port rights are named with positive (unsigned) integers of type *mach_port_name_t*. The type *mach_port_t* is actually the same, but is used when the interface uses the named right as opposed to merely talking about it. Both of these types are of sizeof (void*). When the kernel chooses a name for a new right, it is free to pick any unused name (one that denotes no right) in the space, except for the reserved value MACH_PORT_NULL (0).

There are three basic kinds of rights: receive rights, send rights and send-once rights. A port name can name any of these types of rights, name a port-set, or name nothing.

A port is destroyed, or dies, when its receive right is de-allocated or its ledger is destroyed. When a port dies, rights for the port are not affected, although they can no longer be used to send or receive messages. Any messages queued at the port are destroyed, thereby de-allocating the port rights and out-of-line memory so contained.

Each send-once right generated guarantees the receipt of a single message, either a message sent to that send-once right or, if the send-once right is in any way destroyed, a send-once notification.

A port name has three reference counts. A name is in use when any of the counts is non-zero.

- A receive count. This value is either one or zero, depending on whether the name holds the receive right.
- A user reference (send) count. When a task holds multiple user references for send rights, those rights share a single name. When a task receives a send right that it already holds, the kernel increments the right's user-reference count. When a task de-allocates a send right, the kernel decrements its user-reference count, and the task only loses the send right when the count goes to zero. Since send-once rights are always assigned unique names, send-once rights have a user reference count of one or zero.

  If a task holds both the receive right and send rights for a port, they all share the same name (with their respective reference counts). Send-once rights, however, do not share their name.
- A "dead-name" notification request count. This value is either one or zero, depending on whether a dead-name notification request has been made for the name. The value becomes one when a notification is requested. In becomes zero if the notification is cancelled, or if the notification has been generated (and therefore cannot be cancelled) and the count is explicitly zeroed.

A name that names a port set is considered to have a "port set" reference count of one.

A message can carry port rights; the *msgh_remote* or *msgh_local* fields in the message header or the *disposition* field in a message body descriptor specify the type of port right and how the port right is to be extracted from the caller. The value MACH_PORT_NULL is valid in place of a port right in a message body.

In a sent message, the following *mach_msg_type_name_t* values denote port rights:

MACH_MSG_TYPE_MAKE_SEND
: The message will carry a send right, but the caller must supply a receive right. The send right is created from the receive right.

MACH_MSG_TYPE_COPY_SEND
: The message will carry a send right, and the caller must supply a send right. The user reference count for the supplied send right is not changed.

MACH_MSG_TYPE_MOVE_SEND
: The message will carry a send right, and the caller must supply a send right. The user reference count for the supplied send right is decremented, and the right is destroyed if the count becomes zero. Unless a receive right remains, the name becomes available for recycling.

MACH_MSG_TYPE_MAKE_SEND_ONCE
: The message will carry a send-once right, but the caller must supply a receive right. The send-once right is created from the receive right. Note that send once rights can only be created from the receive right.

MACH_MSG_TYPE_MOVE_SEND_ONCE
: The message will carry a send-once right, and the caller must supply a send-once right. The caller loses the supplied send-once right.

MACH_MSG_TYPE_MOVE_RECEIVE
: The message will carry a receive right, and the caller must supply a receive right. The caller loses the supplied receive right, but retains any send rights with the same name. The port retains attributes like queued messages and extant send and send-once rights.

The following *mach_msg_type_name_t* values in a received message indicate that it carries port rights:

MACH_MSG_TYPE_PORT_SEND
: This value is an alias for MACH_MSG_TYPE_MOVE_SEND. The message carried a send right. If the receiving task already has send and/or receive rights for the port, then that name for the port will be reused. Otherwise, the right will have a new, previously unused, name. If the task already has send rights, it gains a user reference for the right (unless this would cause the user-reference count to overflow). Otherwise, it acquires send rights, with a user-reference count of one.

MACH_MSG_TYPE_PORT_SEND_ONCE
: This value is an alias for MACH_MSG_TYPE_MOVE_SEND_ONCE. The message carried a send-once right. The right will have a new, previously unused, name.

MACH_MSG_TYPE_PORT_RECEIVE
: This value is an alias for MACH_MSG_TYPE_MOVE_RECEIVE. The message carried a receive right. If the receiving task already has send rights for the port, then that name for the port will be reused; otherwise, the right will have a new, previously unused, name.

Memory

A message can carry data "in-line" in its message body. A message can also describe one or more regions of the sender's address space that are to be transferred "out-of-line." In this case, the message is accompanied by a logical copy of the memory.

The sender of the message must explicitly request an out-of-line transfer. Such a region is described as an arbitrary region of the sender's address space. The sender sees this memory as being copied to the receiver.

For each region, the sender has a de-allocate option. If the option is set and the out-of-line memory region is not null, then the region is implicitly de-allocated from the sender, as if by **vm_deallocate**. In particular, the start address is truncated down and the end address rounded up so that every page overlapped by the memory region is de-allocated (thereby possibly de-allocating more memory than is effectively transmitted). The use of this option effectively changes the memory copy into a memory movement. Aside from possibly optimizing the sender's use of memory, the de-allocation option allows the kernel to handle the transfer of memory more efficiently.

Received out-of-line regions are dynamically allocated. Allocated out-of-line memory arrives somewhere in the receiver's address space as new memory. It has the same inheritance and protection attributes as newly **vm_allocate**d memory. The receiver has the responsibility of de-allocating (with **vm_deallocate**) the memory when it is no longer needed. If the message contains more than one region, each will be allocated its own region, not necessarily contiguously.

Null out-of-line memory is legal. If the out-of-line region size is zero, then the region's specified address is ignored. A receive allocated null out-of-line memory region always has a zero address.

Unaligned addresses and region sizes that are not page multiples are legal. A received message can also contain regions with unaligned addresses and sizes that are not multiples of the page size.

Message Send

The send operation queues a message to a port. The message carries a copy of the caller's data. After the send, the caller can freely modify the message buffer or the out-of-line memory regions and the message contents will remain unchanged.

The message carries with it the security ID of the sender in the message header, set by the kernel upon transmission of the message.

Message delivery is reliable and sequenced. Reception of a message guarantees that all messages previously sent to the port by a single task (or a kernel) via send rights have been received and that they are received in the order in which they were sent. Messages sent to send-once rights are unordered.

Message Receive

The receive operation dequeues a message from a port. The receiving task acquires the port rights and out-of-line memory regions carried in the message.

The *rcv_name* argument specifies a port or port set from which to receive. If a port is specified, the caller must possess the receive right for the port and the port must not be a member of a port set. If no message is present, then the call blocks.

If a port set is specified, the call will receive a message sent to any of the member ports. It is permissible for the port set to have no member ports, and ports may be added and removed while a receive from the port set is in progress. The received message can come from any of the member ports that have messages, with the proviso that a member port with messages will not be indefinitely starved. The *msgh_local_port* field in the received message header identifies the port in the port set from which the message was sent.

The *rcv_size* argument specifies the size of the caller's message buffer (which must be big enough for the message header and body); the *msgh_size* field of the received message indicates the actual size of the received message header and body. The **mach_msg** call will not receive a message larger than *rcv_size*. Messages that are too large are destroyed and the function returns an error.

Received messages are stamped with a sequence number, taken from the port from which the message was received. (Messages received from a port set are stamped with a sequence number from the appropriate member port.) Newly created ports start with a zero sequence number. When a message is delivered through a port, it is stamped with the port's sequence number and the port's sequence number is then incremented. Multiple threads receiving messages from a port can use the *msgh_seqno* field to reconstruct the original order of the messages.

The destination and reply ports are reversed in a received message header. The *msgh_local_port* field carries the name of the destination port, from which the message was received, and the *msgh_remote_port* field carries the reply port right. The right description fields are also reversed.

A received message can contain port rights and out-of-line memory. The *msgh_local_port* field does not carry a port right; the act of receiving the message consumes the send or send-once right for the destination port. The *msgh_remote_port* field does carry a port right, and the message can carry additional port rights and memory. Received port rights and memory should be consumed or de-allocated in some fashion.

In almost all cases, *msgh_local_port* will specify the name of a receive right, either *rcv_name*, or, if *rcv_name* is a port set, a member of *rcv_name*. This will be the name the port had at the time the message was dequeued. If some other thread destroys or moves the receive right between the dequeue of the message and its return to the caller, it is possible that the *msgh_local_port* field not name a right the caller holds.

If a resource shortage prevents the reception of a port right, the port right is destroyed and the caller sees the name MACH_PORT_NULL. If a resource shortage prevents the reception of an out-of-line memory region, the region is destroyed and the caller sees a zero address, zero size and the de-allocate flag set in way of a notification.

The following options modify the reception of out-of-line regions:

MACH_RCV_MEM_DEFAULT
: Specifies the default memory reception behavior. (This option can be omitted if the default behavior is desired.) The default behavior assumes that the received memory will be used by the receiver and that the received memory should be pageable temporary memory in the same spirit as that produced by **vm_allocate**.

MACH_RCV_MEM_WIRED
: Specifies that received memory should be wired. This behavior assumes that the receiver will be accessing the memory and expects no unanticipated delay in memory access. The receiver's wired memory ledger limit must be sufficient to permit wiring the memory.

MACH_RCV_PASS_THROUGH
: Specifies that received memory is just passing through the receiver on the way somewhere else. This behavior assumes that the receiver will not be accessing the memory (although accessing the memory is permitted). The memory is temporarily wired until sent in some other message; the receiver's wired memory ledger limit must be sufficient to permit wiring the memory.

RETURN VALUE

The send operation can generate the following return codes. These return codes imply that the call did nothing:

MACH_SEND_MSG_TOO_SMALL
: The specified *send_size* was smaller than the minimum size for a message.

MACH_SEND_INVALID_DATA
: The supplied message buffer was not readable.

MACH_SEND_NO_BUFFER
: A resource shortage prevented allocation of a message buffer.

This return code implies that the message was accepted:

MACH_MSG_SUCCESS
The message was queued.

The receive operation can generate the following return codes. These return codes imply that the call did not dequeue a message. They also indicate that the message was sent, since the send occurs before the receive.

MACH_RCV_INVALID_NAME
The specified *rcv_name* was invalid.

MACH_RCV_IN_SET
The specified port was a member of a port set.

MACH_RCV_PORT_DIED
The caller lost the rights specified by *rcv_name*.

MACH_RCV_PORT_CHANGED
rcv_name specified a receive right that was moved into a port set during the call.

These return codes imply that a message was dequeued and destroyed:

MACH_RCV_INVALID_DATA
The specified message buffer was not writable or too small to receive the header.

These return codes imply that a message was received but the message body could not be completely delivered. The message header is returned to indicate the message size and sequence number, but no rights or out-of-line regions are returned:

MACH_RCV_TOO_LARGE
A message larger than *rcv_size* was dequeued and destroyed.

MACH_RCV_LIMITS
The combined size of all out-of-line memory regions or the total number of port rights in the message exceeds the limit set for the port.

These return codes imply that a message was received:

MACH_MSG_SUCCESS
A message was received.

RELATED INFORMATION

Functions: **vm_allocate**, **vm_deallocate**, **vm_reserve**, **mach_port_request_notification**.

Data Structures: **mach_msg_header**.

CHAPTER 3

Port Manipulation Interface

This chapter discusses the specifics of the kernel's port manipulation interfaces. This includes port, port set and port right related functions. Also included are interfaces that return port related status information that applies to a single task.

ledger_create_port

Function — Create a port drawn from a specified ledger

SYNOPSIS

```
kern_return_t ledger_create_port
        (mach_port_t             target_ledger,
         mach_port_t             target_task,
         mach_port_name_t *      name);
```

DESCRIPTION

The **ledger_create_port** function creates a port whose resources are drawn from a specified ledger. (Ports created by **mach_port_allocate** are drawn from the ledger associated with the target task.)

PARAMETERS

target_ledger
: [in ledger-name send right] The ledger from which to draw.

target_task
: [in task send right] The task into whose space a port should be inserted.

name
: [out scalar] The task's name for the new port. This can be any name that wasn't in use.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_port_allocate**.

mach_port_allocate

Function — Creates a port or port set

SYNOPSIS

kern_return_t **mach_port_allocate**

(mach_port_t	*space,*
mach_port_right_t	*right,*
mach_port_name_t*	*name*);

DESCRIPTION

The **mach_port_allocate** function creates a new right in the specified task. The new right's name is returned in *name*.

PARAMETERS

space
: [in task send right] The task acquiring the port right.

right
: [in scalar] The kind of entity to be created. This is one of the following:

MACH_PORT_RIGHT_RECEIVE
: **mach_port_allocate** creates a port. The new port is not a member of any port set. It doesn't have any extant send or send-once rights. Its sequence number is zero and it has no queued messages. *name* denotes the receive right for the new port.

: *space* does not hold send rights for the new port, only the receive right. **mach_port_insert_right** and **mach_port_extract_right** can be used to convert the receive right into a combined send/receive right.

MACH_PORT_RIGHT_PORT_SET
: **mach_port_allocate** creates a port set. The new port set has no members.

MACH_PORT_RIGHT_DEAD_NAME
: **mach_port_allocate** creates a receive right naming a nonexistent port (as if a port were created and then destroyed). Since no port is created, the right cannot be used for the transmission of messages, but send rights can be generated from it and passed as identifiers in messages.

name

[out scalar] The task's name for the port right. This can be any name that wasn't in use.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_port_insert_right,** **mach_port_extract_right**, **mach_port_mod_refs**, **ledger_create_port**.

mach_port_destroy

Function — Removes a task's rights for a name

SYNOPSIS

```
kern_return_t mach_port_destroy
                (mach_port_t                    space,
                 mach_port_name_t               name);
```

DESCRIPTION

The **mach_port_destroy** function de-allocates all rights denoted by a name. The name becomes immediately available for reuse.

For most purposes, **mach_port_mod_refs** is preferable.

If *name* denotes a port set, then all members of the port set are implicitly removed from the port set.

If *name* denotes a receive right that is a member of a port set, the receive right is implicitly removed from the port set.

If *name* denotes a send-once right, then the destruction of the send-once right produces a send-once notification for the port.

If *name* denotes a send-once, send, and/or receive right, and it has a dead-name request registered, then a port-deleted notification is generated (as opposed to a dead-name notification).

If *name* denotes a receive right, remaining messages queued to the port are destroyed. If any send or send-once rights naming the port have dead-name requests registered, then dead-name notifications are generated for them.

PARAMETERS

space
: [in task send right] The task holding the right.

name
: [in scalar] The task's name for the right.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_allocate,** **mach_port_mod_refs**, **mach_port_request_notification**.

mach_port_extract_right

Function — Extracts a port right from a task

SYNOPSIS

```
kern_return_t mach_port_extract_right
        (mach_port_t                 space,
         mach_port_name_t            name,
         mach_msg_type_name_t        msgt_name,
         mach_port_t*                poly,
         mach_msg_type_name_t*       polyPoly);
```

DESCRIPTION

The **mach_port_extract_right** function extracts a port right from the target task and returns it to the caller as if the task sent the right voluntarily, using *msgt_name* as the disposition for the right. See **mach_msg**.

The returned value of *polyPoly* will be MACH_MSG_TYPE_PORT_SEND if a send right is extracted, MACH_MSG_TYPE_PORT_RECEIVE if a receive right is extracted, and MACH_MSG_TYPE_PORT_SEND_ONCE if a send-once right is extracted.

PARAMETERS

space
: [in task send right] The task holding the port right.

name
: [in scalar] The task's name for the port right.

msgt_name
: [in scalar] IPC type, specifying how the right should be extracted.

poly
: [out random right] The extracted right.

polyPoly
: [out scalar] The type of the extracted right.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted an invalid right (that is, *msgt_name* could not be applied to *name*).

RELATED INFORMATION

Functions: **mach_port_insert_right**, **mach_msg**.

mach_port_get_attributes

Function — Returns information about a port

SYNOPSIS

kern_return_t **mach_port_get_attributes**

(mach_port_t	*space,*
mach_port_name_t	*name,*
mach_port_info_flavor_t	*flavor,*
mach_port_info_t	*port_info_out*,
mach_msg_type_number_t*	*port_info_outCnt*);

DESCRIPTION

The **mach_port_get_attributes** function returns an information structure of type *flavor*. The caller must have a receive right for the port.

PARAMETERS

space
: [in task send right] The task holding a receive right to the port in question.

name
: [in scalar] *space*'s name for the port.

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

MACH_PORT_LIMITS_INFO
: Returns the resource limits for the port. The declaration of this data is found in structure **mach_port_limits**.

MACH_PORT_RECEIVE_STATUS
: Returns random information about the rights and messages associated with the port. The declaration of this data is found in structure **mach_port_status**.

port_info_out
: [out structure] Information about the specified port.

port_info_outCnt
: [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 port name parameter and the port set name in the *mach_port_status* structure returned.

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted a right, but not a receive right.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_set_attributes**.

Data Structures: **mach_port_limits**, **mach_port_status**.

mach_port_get_refs

Function — Retrieves the number of user references for a right

SYNOPSIS

kern_return_t **mach_port_get_refs**

(mach_port_t	*space,*
mach_port_name_t	*name,*
mach_port_right_t	*right,*
mach_port_urefs_t*	*refs*);

DESCRIPTION

The **mach_port_get_refs** function returns the number of user references a task has for a right.

If *name* denotes a right, but not the type of right specified, then zero is returned. Otherwise a positive number of user references is returned. Note a name may simultaneously denote send and receive rights. The number of references for receive rights or send-once rights is always one.

PARAMETERS

space
: [in task send right] The task holding the right.

name
: [in scalar] The task's name for the right.

right
: [in scalar] The type of right / reference being examined: MACH_PORT_RIGHT_SEND, MACH_PORT_RIGHT_RECEIVE, MACH_PORT_RIGHT_SEND_ONCE, MACH_PORT_RIGHT_PORT_SET or MACH_PORT_RIGHT_DEAD_NAME.

refs
: [out scalar] Number of user references.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_mod_refs**.

mach_port_insert_right

Function — Inserts a port right into a task

SYNOPSIS

```
kern_return_t mach_port_insert_right
                (mach_port_t                    space,
                 mach_port_t                     poly,
                 mach_msg_type_name_t        polyPoly,
                 mach_port_name_t*               name);
```

DESCRIPTION

The **mach_port_insert_right** function inserts into *task* the caller's right for a port. The value of *poly* may not be MACH_PORT_NULL.

The argument *polyPoly* specifies a right to be inserted and how that right should be extracted from the caller. It should be a value appropriate for **mach_msg**.

If *polyPoly* is MACH_MSG_TYPE_MAKE_SEND, MACH_MSG_TYPE_MOVE_SEND, or MACH_MSG_TYPE_COPY_SEND, then a send right is inserted. If the target already has send rights, then those send rights gain an additional user reference. Otherwise, the target gains a send right, with a user reference count of one.

If *polyPoly* is MACH_MSG_TYPE_MAKE_SEND_ONCE or MACH_MSG_TYPE_MOVE_SEND_ONCE, then a send-once right is inserted. The target gains a send-once right.

If *polyPoly* is MACH_MSG_TYPE_MOVE_RECEIVE, then a receive right is inserted. The receive right is moved into the target task.

PARAMETERS

space
: [in task send right] The task that gets the caller's right.

poly
: [in random right] The port right.

polyPoly
: [in scalar] IPC type of the sent right; e.g., MACH_MSG_TYPE_COPY_SEND or MACH_MSG_TYPE_MOVE_RECEIVE.

name
: [out scalar] The name assigned for *space* for its new right.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_port_extract_right**, **mach_msg**.

mach_port_mod_refs

Function — Changes the number of user refs for a right

SYNOPSIS

kern_return_t **mach_port_mod_refs**

(mach_port_t	*space,*
mach_port_name_t	*name,*
mach_port_right_t	*right,*
mach_port_delta_t	*delta*);

DESCRIPTION

The **mach_port_mod_refs** function requests that the number of user references a task has for a right be changed. This results in the right being destroyed, if the number of user references is changed to zero.

The *name* should denote the specified right. The number of user references for the right is changed by the amount *delta*, subject to the following restrictions: port sets, receive rights, send-once rights and the dead-name request may only have one user reference. The resulting number of user references can't be negative. If the resulting number of all user references is zero, the effect is to de-allocate the right. There is an implementation-defined maximum number of user references. If the call destroys the right, then the effect is as described for **mach_port_destroy**.

The dead-name request reference can be decremented (to zero) only if the dead-name request has been generated (the target port is dead).

PARAMETERS

space
: [in task send right] The task holding the right.

name
: [in scalar] The task's name for the right.

right
: [in scalar] The type of right / reference being modified: MACH_PORT_RIGHT_SEND, MACH_PORT_RIGHT_RECEIVE, MACH_PORT_RIGHT_SEND_ONCE, MACH_PORT_RIGHT_PORT_SET or MACH_PORT_RIGHT_DEAD_NAME.

delta
: [in scalar] Signed change to the number of user references.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted a right, but not the specified right.

RELATED INFORMATION

Functions: **mach_port_destroy**, **mach_port_get_refs**.

mach_port_move_member

Function — Moves a receive right into/out of a port set

SYNOPSIS

```
kern_return_t mach_port_move_member
        (mach_port_t                space,
         mach_port_name_t           member,
         mach_port_name_t           pset);
```

DESCRIPTION

The **mach_port_move_member** function moves a receive right into a port set. If the receive right is already a member of another port set, it is removed from that set first. If the port set is MACH_PORT_NULL, then the receive right is not put into a port set, but removed from its current port set.

PARAMETERS

space
: [in task send right] The task holding the port set and receive right.

member
: [in scalar] The task's name for the receive right.

pset
: [in scalar] The task's name for the port set.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

KERN_INVALID_NAME
: *member* or *pset* did not denote a right.

KERN_INVALID_RIGHT
: *member* denoted a right, but not a receive right, or *pset* denoted a right, but not a port set.

RELATED INFORMATION

Functions: **mach_port_get_attributes**.

mach_port_request_notification

Function — Request a notification of a port event

SYNOPSIS

kern_return_t **mach_port_request_notification**

(mach_port_t	*space,*
mach_port_name_t	*name,*
mach_msg_id_t	*id,*
mach_port_t	*notify,*
mach_msg_type_name_t	*notifyPoly,*
mach_port_t*	*previous*);

DESCRIPTION

The **mach_port_request_notification** function registers a request for a notification and supplies a send-once right that the notification will use. It is an atomic swap, returning the previously registered send-once right (or MACH_PORT_NULL for none). A notification request may be cancelled by providing MACH_PORT_NULL.

PARAMETERS

space
: [in task send right] The task holding the specified right.

name
: [in scalar] The task's name for the right.

id
: [in scalar] The type of notification; it must be one of the following values:

MACH_NOTIFY_DEAD_NAME
: The call requests a dead-name notification. *name* specifies send, receive, or send-once rights for a port. If the port is destroyed (and the right remains, becoming a dead name), then a dead-name notification that carries the name of the right will be sent to the registered send-once right. If the *name* specifies a port already dead, a dead-name notification is immediately generated.

Requesting a dead-name notification increments the name's dead-name reference count (to one). This count is decremented to zero when the dead-name request is cancelled, or, if the dead-name request is generated prior to attempting to cancel it, by explicitly decrementing the dead-name reference count.

If the name is made available for reuse because of a **mach_port_destroy** or a **mach_port_mod_refs** that deallocates the port, the registered send-once right is used to generate a port-deleted notification instead.

notify

[in notify send-once or receive (to be converted to send-once) right] A send-once right, to which the notification will be sent.

notifyPoly

[in scalar] IPC type of the *notify* right; either MACH_MSG_TYPE_MAKE_SEND_ONCE or MACH_MSG_TYPE_MOVE_SEND_ONCE.

previous

[out notify send-once right] The previously registered send-once right.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

KERN_INVALID_NAME

name did not denote a right.

KERN_INVALID_RIGHT

name denoted an invalid right.

RELATED INFORMATION

Functions: **mach_msg**, **mach_port_get_attributes**.

mach_port_set_attributes

Function — Sets attributes of a port

SYNOPSIS

kern_return_t **mach_port_set_attributes**
(mach_port_t *space,*
mach_port_name_t *name,*
mach_port_info_flavor_t *flavor,*
mach_port_info_t *port_info*,
mach_msg_type_number_t *port_infoCnt*);

DESCRIPTION

The **mach_port_set_attributes** function sets attributes of type *flavor.*

PARAMETERS

space
[in task send right] The task holding a receive right to the port in question.

name
[in scalar] *space*'s name for the port.

flavor
[in scalar] The type of attributes to be set. Valid values are:

MACH_PORT_LIMITS_INFO
Sets resource limits (data passing limits) for the port. The declaration of this data is found in structure **mach_port_limits**.

port_info
[pointer to in structure] Attributes for the specified port.

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

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted a right, but not a receive right.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_get_attributes**.

Data Structures: **mach_port_limits**.

mach_port_type

Function — Return information about a task's port name

SYNOPSIS

```
kern_return_t mach_port_type
        (mach_port_t                space,
         mach_port_name_t           name,
         mach_port_type_t*          ptype);
```

DESCRIPTION

The **mach_port_type** function returns information about *space*'s rights for a specific name in its port name space. The returned *ptype* is a bit-mask indicating what rights *space* holds with this name. The bit-mask is composed of the following bits:

MACH_PORT_TYPE_SEND
: The name denotes send rights.

MACH_PORT_TYPE_RECEIVE
: The name denotes a receive right.

MACH_PORT_TYPE_SEND_ONCE
: The name denotes a send-once right.

MACH_PORT_TYPE_DEAD_NAME
: The port so named has been destroyed.

MACH_PORT_TYPE_DNREQUEST
: A dead-name request has been registered for the right.

MACH_PORT_TYPE_PORT_SET
: The name denotes a port set.

PARAMETERS

space
: [in task send right] The task whose port name space is queried.

name
: [in scalar] The name being queried.

ptype
: [out scalar] The type of the name. Indicates what kind of right the task holds for the port or port set.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_get_attributes**.

mach_reply_port

System Trap— Creates a port for the task

SYNOPSIS

mach_port_t **mach_reply_port**
();

DESCRIPTION

The **mach_reply_port** function creates a new port for the current task and returns the name assigned by the kernel. The kernel records the name in the task's port name space and grants the task receive rights for the port. The new port is not a member of any port set.

This function is an optimized version of **mach_port_allocate** that uses no port references. Its main purpose is to allocate a reply port for the task when the task is starting— namely, before it has any ports to use as reply ports for any IPC based system functions.

PARAMETERS

None

CAUTIONS

Although the created port can be used for any purpose, the implementation may optimize its use as a reply port.

RETURN VALUE

MACH_PORT_NULL
No port was allocated.

[reply receive right]
Any other value indicates success.

RELATED INFORMATION

Functions: **mach_port_allocate**.

CHAPTER 4 Virtual Memory Interface

This chapter discusses the specifics of the kernel's virtual memory interfaces. This includes memory status related functions associated with a single task. Functions that are related to, or used by, external memory managers (pagers) are described in other chapters.

host_page_size

Function — Returns the current page size

SYNOPSIS

```
kern_return_t host_page_size
        (mach_port_t        host,
         vm_size_t*         page_size);
```

DESCRIPTION

The **host_page_size** function returns the page size for the given host.

PARAMETERS

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

page_size
: [out scalar] The host's page size (in bytes).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_host_self**.

vm_allocate

Function — Allocates a region of virtual memory

SYNOPSIS

```
kern_return_t vm_allocate
        (mach_port_t             task,
         vm_address_t*           address,
         vm_size_t               size,
         vm_allocate_t           where);
```

DESCRIPTION

The **vm_allocate** function allocates a region of virtual memory in the specified task's address space. A new region is always zero filled.

If *where* does not include VM_ALLOCATE_FIXED, the returned *address* will be at a kernel chosen page boundary; otherwise, the region starts at the beginning of the virtual page containing *address*.

The specified *address* and *size* are rounded to page boundaries, as follows: the region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory affected may be greater than *size*. Use **host_page_size** to find the current virtual page size.

Initially, there are no access restrictions on any of the pages of the newly allocated region. The initial inheritance value is set to copy.

PARAMETERS

task

[in task send right] The port for the task in whose address space the region is to be allocated.

address

[pointer to in/out scalar] If VM_ALLOCATE_FIXED is specified, this is the starting address for the region; otherwise it is ignored. If the region as specified by the given starting address and size would not lie within the task's un-allocated memory, the kernel does not allocate the region. If allocated, the kernel returns the starting address actually used for the allocated region.

size

[in scalar] The number of bytes to allocate.

where

[in scalar] A bit-wise OR of options affecting the allocation. Valid values are:

VM_ALLOCATE_ANYWHERE

This value may be used as a placeholder in the absence of another value. No check is made to prevent conflicting specifications for the allocated region.

VM_ALLOCATE_FIXED

The kernel allocates the region starting at *address*. By default, the region is allocated from a kernel chosen address. Unless VM_ALLOCATE_RESERVED is also set, the specified address range must not be reserved.

VM_ALLOCATE_RESERVED

Permit the kernel to allocate the region from reserved memory. The kernel does not explicitly try to do so.

VM_ALLOCATE_WIRED

The kernel marks the allocated pages wired.

NOTES

To establish different protections or inheritance for the new region, use the **vm_protect** and **vm_inherit** functions.

A task's address space can contain both explicitly allocated memory and automatically allocated memory. The **vm_allocate** function explicitly allocates memory. The kernel automatically allocates memory to hold out-of-line data passed in a message (and received with **mach_msg**). The kernel allocates memory for the passed data as an integral number of pages.

This interface is machine word length dependent because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS

A portion of the specified range is reserved (unless VM_ALLOCATE_RESERVED is specified); a portion of the specified range is allocated (VM_ALLOCATE_FIXED); or, *address* and *size* fall outside the user task boundary (VM_ADDRESS_MIN, VM_ADDRESS_MAX)

KERN_NO_SPACE

There is not enough space in the task's address space to allocate the new region.

RELATED INFORMATION

Functions: **vm_deallocate**, **vm_inherit**, **vm_protect**, **vm_region**, **host_page_size**, **vm_reserve**, **mach_msg**, **host_info**.

vm_behavior_set

Function — Sets expected data access hints

SYNOPSIS

```
kern_return_t vm_behavior_set
        (mach_port_t          task,
         vm_address_t         address,
         vm_size_t            size,
         vm_behavior_t        new_behavior);
```

DESCRIPTION

The **vm_behavior_set** function informs the kernel of the expected access pattern for a region of memory. The kernel uses this information to bias page-in algorithms.

The specified *address* and *size* are rounded to page boundaries, as follows: the region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory affected may be greater than *size*. Use **host_page_size** to find the current virtual page size.

PARAMETERS

task
: [in task send right] The port for the task in whose address space the memory object behavior is to be set.

address
: [in scalar] The starting address for the memory region.

size
: [in scalar] The number of bytes in the region.

new_behavior
: [in scalar] The expected reference pattern for the memory. Possible values are:

  VM_BEHAVIOR_DEFAULT
  : The system's default behavior. Assumes strong locality of reference.

  VM_BEHAVIOR_RANDOM
  : No particular order expected. Assumes weak locality of reference.

VM_BEHAVIOR_FSEQNTL
: Forward sequential order. Learning of this behavior, the kernel may (forward) pre-fetch pages; this way, future page faults can be satisfied more quickly.

VM_BEHAVIOR_RSEQNTL
: Reverse sequential order. Learning of this behavior, the kernel may (reverse) pre-fetch pages; this way, future page faults can be satisfied more quickly.

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS
: No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

RELATED INFORMATION

Functions: **vm_region**, **host_page_size**, **host_info**.

vm_copy

Function — Copies a region in a task's virtual memory

SYNOPSIS

```
kern_return_t vm_copy
        (mach_port_t          task,
         vm_address_t         source_address,
         vm_size_t            size,
         vm_address_t         dest_address);
```

DESCRIPTION

The **vm_copy** function copies a source region to a destination region within the same task's virtual memory. It is semantically equivalent to **vm_read** followed by **vm_write**. The destination region can overlap the source region.

The destination region must already be allocated. The source region must be readable, and the destination region must be writable.

PARAMETERS

task
: [in task send right] The port for the task whose memory is to be copied.

source_address
: [in scalar] The starting address for the source region. The address may be anywhere within an allocated page.

size
: [in scalar] The number of bytes in the source region. The number of bytes must convert to an integral number of virtual pages.

dest_address
: [in scalar] The starting address for the destination region. The address may be anywhere within an allocated page.

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS
: No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

KERN_PROTECTION_FAILURE
: The source region is protected against reading, or the destination region is protected against writing.

KERN_MEMORY_ERROR
: Some portion of the range is un-allocated or inaccessible.

RELATED INFORMATION

Functions: **vm_protect**, **vm_read**, **vm_write**, **host_page_size**, **host_info**.

vm_deallocate

Function — Deallocates a region of virtual memory

SYNOPSIS

```
kern_return_t vm_deallocate
        (mach_port_t        task,
         vm_address_t       address,
         vm_size_t          size);
```

DESCRIPTION

The **vm_deallocate** function deallocates a region of virtual memory in the specified task's address space. If the specified range is reserved, the space is deallocated but continues to be reserved.

The specified *address* and *size* are rounded to page boundaries, as follows: the region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory affected may be greater than *size*. Use **host_page_size** to find the current virtual page size.

vm_deallocate affects only *task*. Other tasks that have access to the deallocated memory can continue to reference it.

PARAMETERS

task
: [in task send right] The port for the task in whose address space the region is to be deallocated.

address
: [in scalar] The starting address for the region.

size
: [in scalar] The number of bytes to deallocate.

NOTES

vm_deallocate can be used to deallocate memory passed as out-of-line data in a message.

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS
: No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

RELATED INFORMATION

Functions: **mach_msg**, **vm_allocate**, **vm_map**, **host_page_size**, **host_info**.

vm_inherit

Function — Sets the inheritance attribute for a region of virtual memory

SYNOPSIS

kern_return_t **vm_inherit**
(mach_port_t *task,*
vm_address_t *address,*
vm_size_t *size,*
vm_inherit_t *new_inheritance*);

DESCRIPTION

The **vm_inherit** function sets the inheritance attribute for a region within the specified task's address space. The inheritance attribute determines the type of access established for child tasks at task creation.

The region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory deallocated may be greater than *size*. Use **host_page_size** to find the current virtual page size.

A parent and a child task can share the same physical memory only if the inheritance for the memory is set to VM_INHERIT_SHARE before the child task is created. Other than through the use of an external memory manager (see **vm_map**), this is the only way that two tasks can share memory.

Note that all the threads within a task share the task's memory.

PARAMETERS

task
[in task send right] The port for the task whose address space contains the region.

address
[in scalar] The starting address for the region.

size
[in scalar] The number of bytes in the region.

new_inheritance
[in scalar] The new inheritance attribute for the region. Valid values are:

VM_INHERIT_SHARE
Allows child tasks to share the region.

VM_INHERIT_COPY
: Gives child tasks a copy of the region.

VM_INHERIT_NONE
: Provides no access to the region for child tasks.

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS
: No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

RELATED INFORMATION

Functions: **task_create**, **vm_map**, **vm_region**, **vm_reserve**, **host_info**.

vm_map

Function — Maps a memory object to a task's address space

SYNOPSIS

kern_return_t **vm_map**	
(mach_port_t	*task,*
vm_address_t*	*address,*
vm_size_t	*size,*
vm_address_t	*mask,*
vm_allocate_t	*where,*
mach_port_t	*memory_object,*
vm_offset_t	*offset,*
boolean_t	*copy,*
vm_prot_t	*cur_protection,*
vm_prot_t	*max_protection,*
vm_inherit_t	*inheritance*);

DESCRIPTION

The **vm_map** function maps a portion of the specified memory object into the virtual address space belonging to *task*. The target task can be the calling task or another task, identified by its task kernel port.

The portion of the memory object mapped is determined by *offset* and *size*. The kernel maps *address* to the offset, so that an access to the memory starts at the offset in the object.

The region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory deallocated may be greater than *size*. Use **host_page_size** to find the current virtual page size.

The *mask* parameter specifies additional alignment restrictions on the kernel's selection of the starting address. Uses for this mask include:

- Forcing the memory address alignment for a mapping to be the same as the alignment within the memory object.
- Quickly finding the beginning of an allocated region by performing bit arithmetic on an address known to be in the region.
- Emulating a larger virtual page size.

The *cur_protection*, *max_protection*, and *inheritance* parameters set the protection and inheritance attributes for the mapped object. As a rule, at least the maximum protection should be specified so that a server can make a restricted (for example, read-only) mapping in a client atomically. The current protection and inheritance parameters are provided for convenience so that the caller does not have to call **vm_inherit** and **vm_protect** separately.

The same memory object can be mapped more than once and by more than one task. If an object is mapped by multiple tasks, the kernel maintains consistency for all the mappings if they are on the same host. In this case, the virtual memory to which the object is mapped is shared by all the tasks. Changes made by one task in its address space are visible to all the other tasks.

If the *memory_object* is known to the kernel on which *task* is executing then the kernel knows the identity of the underlying memory object and as a result the attributes of that object as well as the maximum access allowed by this representative. If the representative is not known, the kernel will engage in a dialog with the memory manager holding the receive right to the representative port (see **memory_object_notify**) from which the kernel will obtain these attributes. These attributes are necessary to determine the validity of the **vm_map** call.

PARAMETERS

task

[in task send right] The port for the task in whose address space the memory object is to be mapped.

address

[pointer to in/out scalar] If VM_ALLOCATE_FIXED is specified, this is the starting address for the region; otherwise it is the address from which a search for free space begins. If there is not enough room following the address, the kernel does not map the object. The kernel returns the starting address actually used for the mapped object.

size

[in scalar] The number of bytes to allocate for the object. The kernel rounds this number up to an integral number of virtual pages.

mask

[in scalar] Alignment restrictions for starting address. Bits turned on in the mask will not be turned on in the starting address. The mask must be $\mathbf{2}^{n}\ \mathbf{-\ 1}$ for some *n*.

where

[in scalar] Bit-wise OR of various placement options. Valid values are:

VM_ALLOCATE_FIXED

The kernel maps the region starting at *address*. By default, the region is mapped at a kernel chosen address. Unless VM_ALLOCATE_RESERVED is also set, the specified address range must not be reserved.

VM_ALLOCATE_RESERVED

Permit the kernel to allocate the region from reserved memory. The kernel does not explicitly try to do so.

VM_ALLOCATE_ANYWHERE
: This value may be used as a placeholder in the absence of another value. No check is made to prevent conflicting specifications for the allocated region.

memory_object
: [in memory-object-representative send right] The port naming the memory object representative. If MEMORY_OBJECT_NULL is specified, the kernel allocates zero-filled memory, as with **vm_allocate**.

offset
: [in scalar] An offset within the memory object, in bytes. The kernel maps *address* to the specified offset. The offset must be on a page boundary.

copy
: [in boolean] Copy indicator. If true, the kernel copies the region of the memory object to the specified task's address space. If false, the region is directly mapped.

cur_protection
: [in scalar] The initial current protection for the region. Valid values are obtained by or'ing together the following values:

VM_PROT_READ
: Allows read access.

VM_PROT_WRITE
: Allows write access.

VM_PROT_EXECUTE
: Allows execute access.

max_protection
: [in scalar] The maximum protection for the region. Values are the same as for *cur_protection*.

inheritance
: [in scalar] The initial inheritance attribute for the region. Valid values are:

VM_INHERIT_SHARE
: Allows child tasks to share the region.

VM_INHERIT_COPY
: Gives child tasks a copy of the region.

VM_INHERIT_NONE
: Provides no access to the region for child tasks.

NOTES

vm_map allocates a region in a task's address space and maps the specified memory object to this region. **vm_allocate** allocates a zero-filled temporary region in a task's address space.

Before a memory object can be mapped, a port naming it must be acquired from the memory manager serving it.

This interface is machine word length specific because of the virtual address parameter.

CAUTIONS

Do not attempt to map a memory object unless it has been provided by a memory manager that implements the memory object interface. If another type of port is specified, a thread that accesses the mapped virtual memory may become permanently hung or may receive a memory exception.

RETURN VALUE

KERN_INVALID_ADDRESS
: A portion of the specified range is reserved (unless VM_ALLOCATE_RESERVED is specified); a portion of the specified range is allocated (VM_ALLOCATE_FIXED); or, *address* and *size* fall outside the user task boundary (VM_ADDRESS_MIN, VM_ADDRESS_MAX)

KERN_NO_SPACE
: There is not enough space in the task's address space to allocate the new region for the memory object.

KERN_PROTECTION_FAILURE
: *max_protection* or *cur_protection* exceeds that permitted by *memory_object*.

RELATED INFORMATION

Functions: **memory_object_notify**, **vm_allocate**, **vm_reserve**, **host_info**.

vm_protect

Function — Sets access privileges for a region of virtual memory

SYNOPSIS

kern_return_t **vm_protect**

(mach_port_t	*task,*
vm_address_t	*address,*
vm_size_t	*size,*
boolean_t	*set_maximum,*
vm_prot_t	*new_protection*);

DESCRIPTION

The **vm_protect** function sets access privileges for a region within the specified task's address space. *new_protection* specifies a combination of read, write, and execute accesses that are allowed (rather than prohibited).

The region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory deallocated may be greater than *size*. Use **host_page_size** to find the current virtual page size.

The enforcement of virtual memory protection is machine-dependent. Nominally read access requires VM_PROT_READ permission, write access requires VM_PROT_WRITE permission, and execute access requires VM_PROT_EXECUTE permission. However, some combinations of access rights may not be supported. In particular, the kernel interface allows write access to require VM_PROT_READ and VM_PROT_WRITE permission and execute access to require VM_PROT_READ permission.

Protection cannot be set for regions reserved but not allocated.

PARAMETERS

task
: [in task send right] The port for the task whose address space contains the region.

address
: [in scalar] The starting address for the region.

size
: [in scalar] The number of bytes in the region.

set_maximum
[in boolean] Maximum/current indicator. If true, the new protection sets the maximum protection for the region. If false, the new protection sets the current protection for the region. If the maximum protection is set below the current protection, the current protection is reset to the new maximum.

new_protection
[in scalar] The new protection for the region. Valid values are obtained by ORing together the following values:

VM_PROT_READ
Allows read access.

VM_PROT_WRITE
Allows write access.

VM_PROT_EXECUTE
Allows execute access.

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_PROTECTION_FAILURE
The new protection increased the current or maximum protection beyond the existing maximum protection.

KERN_INVALID_ADDRESS
No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

RELATED INFORMATION

Functions: **vm_inherit**, **vm_region**, **host_info**.

vm_read

Function — Reads a task's virtual memory

SYNOPSIS

```
kern_return_t vm_read
        (mach_port_t                    task,
         vm_address_t                   address,
         vm_size_t                      size,
         pointer_t*                     data,
         mach_msg_type_number_t*        dataCnt);
```

DESCRIPTION

The **vm_read** function reads a portion of a task's virtual memory. It allows one task to read another task's memory.

PARAMETERS

task
: [in task send right] The port for the task whose memory is to be read.

address
: [in scalar] The address at which to start the read. This address may be anywhere within an allocated page.

size
: [in scalar] The number of bytes to read.

data
: [out pointer to dynamic array of bytes] The array of data returned by the read.

dataCnt
: [out scalar] The size returned (in natural-sized units).

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS
: No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

KERN_PROTECTION_FAILURE
Specified memory is valid, but does not permit reading.

KERN_MEMORY_ERROR
Some portion of the range is unallocated or inaccessible.

RELATED INFORMATION

Functions: **vm_read_inline**, **vm_copy**, **vm_deallocate**, **vm_write**, **host_info**.

vm_read_inline

System Trap — Reads a task's virtual memory

SYNOPSIS

kern_return_t **vm_read_inline**

(mach_port_t	*task,*
vm_address_t	*address,*
vm_size_t	*size,*
void*	*ubuf*);

DESCRIPTION

The **vm_read_inline** function reads a portion of a task's virtual memory. It allows one task to read another task's memory.

PARAMETERS

task
: [in task send right] The port for the task whose memory is to be read.

address
: [in scalar] The address at which to start the read.

size
: [in scalar] The number of bytes to read.

ubuf
: [pointer to out array of bytes] The array of data returned by the read.

NOTES

This interface is machine word length specific because of the virtual address parameter.

This interface differs from **vm_read** in that **vm_read** transfers data out-of-line, thereby permitting larger transfers but at greater cost.

RETURN VALUE

KERN_INVALID_ADDRESS
: No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

KERN_PROTECTION_FAILURE
: Specified memory is valid, but does not permit reading.

KERN_MEMORY_ERROR
Some portion of the range is un-allocated or inaccessible.

RELATED INFORMATION

Functions: **vm_read**, **vm_copy**, **vm_write**, **host_info**.

vm_region

Function — Returns information on a region of virtual memory

SYNOPSIS

kern_return_t **vm_region**

(mach_port_t	*target_task,*
vm_address_t*	*address,*
vm_size_t*	*size,*
vm_region_flavor_t	*flavor,*
vm_region_basic_info_t	*info,*
mach_msg_type_number_t*	*infoCnt,*
mach_port_t*	*representative*);

DESCRIPTION

The **vm_region** function returns information on a region within the specified task's address space.

The function begins looking at *address* and continues until it finds an allocated region. If the input address is within a region, the function uses the start of that region. The starting address for the located region is returned in *address*.

PARAMETERS

target_task
: [in task send right] The port for the task whose address space contains the region.

address
: [pointer to in/out scalar] The address at which to start looking for a region. The function returns the starting address actually used.

size
: [out scalar] The number of bytes in the located region. The number converts to an integral number of virtual pages.

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

VM_REGION_BASIC_INFO
: Basic information about the region (size, inheritance, etc.). This information is declared by **vm_region_basic_info**.

info
: [out structure] Returned region information.

infoCnt

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

representative

[out memory-object-representative send right] The name of a send right to the memory object representative port through which the underlying memory object is to be accessed. This value can be used as a name to identify regions mapped against the same memory object. This value will be null for anonymous memory regions, virtually copied memory, or memory inherited from a parent task (that is, for all cases except memory directly mapped in this task with **vm_map**).

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS

There is no region at or beyond the specified starting address.

RELATED INFORMATION

Functions: **vm_allocate**, **vm_deallocate**, **vm_inherit**, **vm_protect**, **vm_behavior_set**, **vm_reserve**, **memory_object_notify**, **host_info**.

Data Structures: **vm_region_basic_info**

vm_reserve

Function — Marks a memory region as reserved

SYNOPSIS

```
kern_return_t vm_reserve
        (mach_port_t        task,
         vm_address_t       address,
         vm_size_t          size,
         boolean_t          inherit);
```

DESCRIPTION

The **vm_reserve** function marks the specified region as reserved. A reserved region is one that will not be used for random allocation (out-of-line memory reception or non-explicit memory allocation).

PARAMETERS

task
: [in task send right] The port for the task whose address space contains the region.

address
: [in scalar] The starting address for the region.

size
: [in scalar] The number of bytes in the region.

inherit
: [in boolean] If true, the reservation is propagated to child tasks created with **task_create** (with the inheritance parameter set to true). Otherwise, the range is not reserved in child tasks.

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS
: A reserved region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

RELATED INFORMATION

Functions: **vm_allocate**, **vm_deallocate**, **mach_msg**, **vm_map**, **vm_region**, **vm_inherit**, **vm_unreserve**, **host_info**.

vm_unreserve

Function — Marks a memory region as no longer reserved

SYNOPSIS

```
kern_return_t vm_unreserve
        (mach_port_t        task,
         vm_address_t       address,
         vm_size_t          size);
```

DESCRIPTION

The **vm_unreserve** function marks the specified region as no longer reserved. A reserved region is one that will not be used for random allocation (out-of-line memory reception or non-explicit memory allocation).

PARAMETERS

task
[in task send right] The port for the task whose address space contains the region.

address
[in scalar] The starting address for the region.

size
[in scalar] The number of bytes in the region.

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS
No reserved region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

RELATED INFORMATION

Functions: **vm_allocate**, **vm_deallocate**, **mach_msg**, **vm_map**, **vm_region**, **vm_inherit**, **vm_reserve**, **host_info**.

vm_unwire

Function — Resets the pageability of a region of virtual memory

SYNOPSIS

kern_return_t **vm_unwire**
(mach_port_t *task,*
vm_address_t *address,*
vm_size_t *size*);

DESCRIPTION

The **vm_unwire** function clears the unpageable flag ("wired" flag) for a region within the specified task's address space.

PARAMETERS

task
[in task send right] The port for the task whose address space contains the region.

address
[in scalar] The starting address for the region.

size
[in scalar] The number of bytes in the region.

NOTES

This interface is machine word length specific because of the virtual address parameter.

Wired memory is counted against the task's wired memory ledger limit.

RETURN VALUE

KERN_INVALID_ADDRESS
No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

RELATED INFORMATION

Functions: **thread_wire**, **vm_wire**, **host_info**.

vm_wire

Function — Forces pages to be and stay resident

SYNOPSIS

```
kern_return_t vm_wire
        (mach_port_t        task,
         vm_address_t       address,
         vm_size_t          size);
```

DESCRIPTION

The **vm_wire** function sets the unpageable flag ("wired" flag) for a region within the specified task's address space. A page is wired into physical memory if any task accessing it has wired it. Once wired, the wiring task will take no memory faults against the page that would result in unpredictable access time to the page.

PARAMETERS

task
: [in task send right] The port for the task whose address space contains the region.

address
: [in scalar] The starting address for the region.

size
: [in scalar] The number of bytes in the region.

NOTES

This interface is machine word length specific because of the virtual address parameter.

The wired memory is counted against the task's wired memory ledger limit.

RETURN VALUE

KERN_INVALID_ADDRESS
: No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

KERN_MEMORY_ERROR
: Some portion of the range is un-allocated or inaccessible.

RELATED INFORMATION

Functions: **thread_wire**, **vm_unwire**, **host_info**.

vm_write

Function — Writes data to a task's virtual memory

SYNOPSIS

```
kern_return_t vm_write
        (mach_port_t                     task,
        vm_address_t                     address,
        pointer_t                        data,
        mach_msg_type_number_t           dataCnt);
```

DESCRIPTION

The **vm_write** function writes an array of data to a task's virtual memory. It allows one task to write to another task's memory.

The result of **vm_write** is as if *task* had directly written into the set of pages. Hence, *task* must have write permission to the pages.

PARAMETERS

task
: [in task send right] The port for the task whose memory is to be written.

address
: [in scalar] The address at which to start the write. The starting address may be anywhere within an allocated page.

data
: [pointer to array of bytes] An array of data to be written.

dataCnt
: [in scalar] The number of bytes in the array.

NOTES

This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

KERN_INVALID_ADDRESS
: No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

KERN_PROTECTION_FAILURE
Specified memory is valid, but does not permit writing.

KERN_MEMORY_ERROR
Some portion of the range is un-allocated or inaccessible.

RELATED INFORMATION

Functions: **vm_write_inline**, **vm_copy**, **vm_protect**, **vm_read**, **host_page_size**, **host_info**.

vm_write_inline

System Trap — Writes data to a task's virtual memory

SYNOPSIS

kern_return_t **vm_write_inline**

(mach_port_t	*task,*
vm_address_t	*address,*
void*	*ubuf,*
vm_size_t	*size*);

DESCRIPTION

The **vm_write_inline** function writes an array of data to a task's virtual memory. It allows one task to write to another task's memory.

The result of **vm_write_inline** is as if *task* had directly written into the set of pages. Hence, *task* must have write permission to the pages.

PARAMETERS

task
: [in task send right] The port for the task whose memory is to be written.

address
: [in scalar] The address at which to start the write.

ubuf
: [pointer to in array of bytes] The array of data to write.

size
: [in scalar] The number of bytes in the array. There is no explicit limit, but a sufficiently large size yields KERN_RESOURCE_LIMIT.

NOTES

This interface is machine word length specific because of the virtual address parameter.

This interface differs from **vm_write** in that **vm_write** transfers data out-of-line, thereby permitting larger transfers but at greater cost.

RETURN VALUE

KERN_INVALID_ADDRESS
No allocated region exists within the specified range, or the range itself is invalid (that is, it exceeds the bounds of VM_ADDRESS_MIN and VM_ADDRESS_MAX).

KERN_PROTECTION_FAILURE
Specified memory is valid, but does not permit writing.

KERN_MEMORY_ERROR
Some portion of the range is un-allocated or inaccessible.

RELATED INFORMATION

Functions: **vm_write**, **vm_copy**, **vm_protect**, **vm_read**, **host_info**.

CHAPTER 5 External Memory Management Functions

This chapter discusses the interfaces used by external memory managers when invoking the kernel.

memory_object_change_attributes

Function — Changes various object attributes

SYNOPSIS

```
kern_return_t memory_object_change_attributes
        (mach_port_t                 mem_cache_control,
         memory_object_flavor_t      flavor,
         memory_object_info_t        attributes,
         mach_msg_type_number_t      attributesCnt,
         mach_port_t                 reply_port);
```

DESCRIPTION

The **memory_object_change_attributes** function sets various attributes of the specified memory object.

PARAMETERS

mem_cache_control
: [in memory-cache-control send right] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_notify** call.

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

MEMORY_OBJECT_PERFORMANCE_INFO
: Performance related attributes such as the cache indicator and the cluster size. *attributes* should specify a structure of type **memory_object_perf_info**.

MEMORY_OBJECT_BEHAVIOR_INFO
: Behavior related attributes such as the copy strategy and sync invalidate flag. *attributes* should specify a structure of type **memory_object_behave_info**.

attributes
: [pointer to in structure] New attributes.

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

reply_port
: [in reply receive (to be converted to send) right] A port to which a reply (**memory_object_change_completed**) is to be sent indicating the completion of the attribute change; may be MACH_PORT_NULL.

RETURN VALUE

Since this function is asynchronous, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_change_completed**, **memory_object_get_attributes**, **memory_object_notify**.

Data Structures: **memory_object_perf_info**, **memory_object_behave_info**.

memory_object_data_error

Function — Indicates no data for a memory object

SYNOPSIS

kern_return_t **memory_object_data_error**

(mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
vm_size_t	*size,*
kern_return_t	*error_value*);

DESCRIPTION

The **memory_object_data_error** function indicates that the memory manager cannot provide the kernel with the data requested for the given region, specifying a reason for the error.

When the kernel issues a **memory_object_data_request** call, the memory manager can respond with a **memory_object_data_error** call to indicate that the page cannot be retrieved, and that a memory failure exception should be raised in any client threads that are waiting for the page. Clients are permitted to catch these exceptions and retry their page faults. As a result, this call can be used to report transient errors as well as permanent ones. A memory manager can use this call for both hardware errors (for example, disk failures) and software errors (for example, accessing data that does not exist or is protected).

PARAMETERS

mem_cache_control
: [in memory-cache send right] The memory cache port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_notify** call.

offset
: [in scalar] The offset within the memory object, in bytes. The offset must be on a page boundary.

size
: [in scalar] The number of bytes of data (starting at *offset*). The number must convert to an integral number of memory object pages.

error_value
: [in scalar] Reason for the error.

NOTES

If *error_value* has a system code of *err_kern*, the kernel will substitute an error value of KERN_MEMORY_ERROR.

RETURN VALUE

Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_data_request**, **memory_object_data_supply**, **memory_object_data_unavailable**.

memory_object_data_supply

Function — Supplies data for a region of a memory object

SYNOPSIS

kern_return_t **memory_object_data_supply**

(mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
pointer_t	*data,*
mach_msg_type_number_t	*dataCnt,*
boolean_t	*dataDealloc*,
vm_prot_t	*lock_value,*
boolean_t	*precious*,
mach_port_t	*reply_port*);

DESCRIPTION

The **memory_object_data_supply** function supplies the kernel with a range of data for the specified memory object. A memory manager can only provide data that was requested by a **memory_object_data_request** call from the kernel.

PARAMETERS

mem_cache_control
: [in memory-cache send right] The memory cache port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_notify** call.

offset
: [in scalar] The offset within the memory object, in bytes. The offset must be on a page boundary.

data
: [pointer to page aligned in array of bytes] The address of the data being provided to the kernel.

dataCnt
: [in scalar] The number of bytes of data to be provided. The number must be an integral number of memory object pages.

dataDealloc
: [in scalar] If TRUE, the pages to be copied (starting at *data*) will be deallocated from the memory manager's address space as a result of being copied into the message, allowing the pages to be moved into the kernel instead of being physically copied.

lock_value

[in scalar] One or more forms of access **not** permitted for the specified data. Valid values are:

VM_PROT_NONE

Prohibits no access (that is, all forms of access are permitted).

VM_PROT_ALL

Prohibits all forms of access.

Other valid values are obtained by ORing together the following values:

VM_PROT_READ

Prohibits read access.

VM_PROT_WRITE

Prohibits write access.

VM_PROT_EXECUTE

Prohibits execute access.

precious

[in boolean] If TRUE, the pages being supplied are “precious”, that is, the memory manager is not (necessarily) retaining its own copy. These pages must be returned to the manager when evicted from memory, even if not modified.

reply_port

[in reply receive (to be converted to send) right] A port to which the kernel should send a **memory_object_supply_completed** to indicate the status of the accepted data. MACH_PORT_NULL is allowed. The reply message indicates which pages have been accepted.

NOTES

The kernel accepts only integral numbers of pages. It discards any partial pages without notification.

CAUTIONS

A memory manager must be careful that it does not attempt to provide data that has not been explicitly requested. In particular, a memory manager must ensure that it does not provide writable data again before it receives back modifications from the kernel. This may require that the memory manager remember which pages it has provided, or that it exercise other cache control functions (via **memory_object_lock_request**) before proceeding. The kernel prohibits the overwriting of live data pages and will not accept pages it has not requested.

RETURN VALUE

Since this function is asynchronous, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_data_error**, **memory_object_data_request**, **memory_object_data_unavailable**, **memory_object_lock_request**, **memory_object_supply_completed**.

memory_object_data_unavailable

Function — Indicates no data for a memory object

SYNOPSIS

```
kern_return_t memory_object_data_unavailable
        (mach_port_t                mem_cache_control,
        vm_offset_t                 offset,
        vm_size_t                   size);
```

DESCRIPTION

The **memory_object_data_unavailable** function indicates that the memory manager cannot provide the kernel with the data requested for the given region. Instead, the kernel should provide the data (zero-filled pages) for this region.

PARAMETERS

mem_cache_control
: [in memory-cache-control send right] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_notify** call.

offset
: [in scalar] The offset within the memory object, in bytes. The offset must be on a page boundary.

size
: [in scalar] The number of bytes of data (starting at *offset*). The number must convert to an integral number of memory object pages.

RETURN VALUE

Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_data_error**, **memory_object_data_request**, **memory_object_data_supply**.

memory_object_destroy

Function — Shuts down a memory object

SYNOPSIS

kern_return_t **memory_object_destroy**
(mach_port_t *mem_cache_control,*
kern_return_t *reason*);

DESCRIPTION

The **memory_object_destroy** function tells the kernel to shut down the specified memory object. As a result of this call, the kernel no longer supports paging activity or any memory object calls on the memory object. The kernel issues a **memory_object_terminate** call.

To ensure that any modified cached data is returned before the object is terminated, the memory manager should call **memory_object_lock_request** with *should_flush* set and a lock value of VM_PROT_WRITE before it makes the **memory_object_destroy** call.

PARAMETERS

mem_cache_control
: [in memory-cache-control send right] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_notify** call.

reason
: [in scalar] An error code indicating when the object must be destroyed.

NOTES

The *reason* code is currently ignored by the kernel.

RETURN VALUE

Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_lock_request**, **memory_object_terminate**.

memory_object_establish

Function — Marks a memory object ready to receive paging operations

SYNOPSIS

```
kern_return_t memory_object_establish
        (mach_port_t                    host_name,
         mach_port_t                    ledger,
         mach_port_t                    mem_cache_control,
         mach_port_t                    representative,
         mach_port_t                    abs_rep,
         vm_prot_t                      rep_max_prot,
         security_id_t                  security_id,
         memory_object_flavor_t         flavor,
         memory_object_info_t           behavior,
         mach_msg_type_number_t         behaviorCnt);
```

DESCRIPTION

The **memory_object_establish** function informs the kernel of the identity of the abstract memory object (port) underlying a given memory object representative. This call is used (as opposed to **memory_object_representative**) when it is known that the underlying memory object is not initialized on the given host. The call supplies the attributes of the memory cache object to be created as well as the maximum protection implied by the representative. This call is the memory manager's reply to a previous **memory_object_notify** call. After this call, the manager must be prepared to handle the object.

If the kernel was not expecting this message, the kernel responds with **memory_object_rejected**; otherwise, the kernel sends no response until the kernel needs servicing by the manager.

PARAMETERS

host_name
: [in host-name send right] The name (or control) port for the host upon which the object is being mapped. The name port is provided by the kernel in the **memory_object_notify** call.

ledger
: [in ledger-name send right] Resource ledger (on the destination host) from which the memory object will draw its resources.

mem_cache_control
: [in memory-cache-control send right] The control port to the memory object provided by the kernel in the **memory_object_notify** call.

representative
: [in memory-object-representative receive (to be converted to send) right] The memory object representative port from the corresponding **memory_object_notify** message.

abs_rep
: [in abstract-memory-object receive (to be converted to send) right] The underlying abstract memory object. All subsequent messages from the kernel will come over this port.

rep_max_prot
: [in scalar] The maximum protections that can be specified by any task mapping this object via the memory object representative port over which the initial **memory_object_notify** message arrived.

security_id
: [in scalar] The security ID that tasks attempting to map this representative must have. A zero value indicates all identities.

flavor
: [in scalar] Type of the behavior attribute set.

MEMORY_OBJECT_BEHAVIOR_INFO
: Behavior related attributes such as the copy strategy and sync invalidate flag. *behavior* should specify a structure of type **memory_object_behave_info**.

behavior
: [pointer to in structure] Behavior attribute set for the memory object.

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

RETURN VALUE

Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_change_attributes**, **memory_object_get_attributes**, **memory_object_notify**, **memory_object_representative**, **memory_object_rejected**.

Data Structures: **memory_object_behave_info**.

memory_object_get_attributes

Function — Returns current attributes for a memory object

SYNOPSIS

kern_return_t **memory_object_get_attributes**
(mach_port_t *mem_cache_control,*
memory_object_flavor_t *flavor*,
memory_object_info_t *attributes*,
mach_msg_type_number_t* *attributesCnt*);

DESCRIPTION

The **memory_object_get_attributes** function retrieves the current attributes for the specified memory object.

PARAMETERS

mem_cache_control
[in memory-cache-control send right] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_notify** call.

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

MEMORY_OBJECT_PERFORMANCE_INFO
Performance related attributes such as the cache indicator and the cluster size. *attributes* should specify a structure of type **memory_object_perf_info**.

MEMORY_OBJECT_BEHAVIOR_INFO
Behavior related attributes such as the copy strategy and sync invalidate flag. *attributes* should specify a structure of type **memory_object_behave_info**.

attributes
[out structure] Current attributes.

attributesCnt
[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: **memory_object_change_attributes**, **memory_object_establish**.

Data Structures: **memory_object_perf_info**, **memory_object_behave_info**.

memory_object_lock_request

Function — Restricts access to memory object data

SYNOPSIS

```
kern_return_t memory_object_lock_request
        (mach_port_t                mem_cache_control,
         vm_offset_t                offset,
         vm_size_t                  size,
         memory_object_return_t     should_return,
         boolean_t                  should_flush,
         vm_prot_t                  lock_value,
         mach_port_t                reply_to);
```

DESCRIPTION

The **memory_object_lock_request** function allows the memory manager to make the following requests of the kernel:

- Clean the pages within the specified range by writing back all changed (that is, dirty) and precious pages. The kernel uses the **memory_object_data_return** call to write back the data. The *should_return* parameter must be set to non-zero.
- Flush all cached data within the specified range. The kernel invalidates the range of data and revokes all uses of that data. The *should_flush* parameter must be set to true.
- Alter access restrictions specified in the **memory_object_data_supply** call or a previous **memory_object_lock_request** call. The *lock_value* parameter must specify the new access restrictions. Note that this parameter can be used to unlock previously locked data.

Once the kernel performs all of the actions requested by this call, it issues a **memory_object_lock_completed** call using the *reply_to* port.

PARAMETERS

mem_cache_control
: [in memory-cache-control send right] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_notify** call.

offset
: [in scalar] The offset within the memory object, in bytes. The offset must be on a page boundary.

size

[in scalar] The number of bytes of data (starting at *offset*) to be affected. The number must convert to an integral number of memory object pages.

should_return

[in scalar] Clean indicator. Values are:

MEMORY_OBJECT_RETURN_NONE

Don't return any pages. If *should_flush* is TRUE, pages will be discarded.

MEMORY_OBJECT_RETURN_DIRTY

Return only dirty (modified) pages. This is valid only if *should_flush* is FALSE. The kernel returns modified pages and maintains responsibility for precious pages.

MEMORY_OBJECT_RETURN_ALL

Both dirty and precious pages are returned. If *should_flush* is FALSE, the kernel maintains responsibility for the precious pages.

should_flush

[in boolean] Flush indicator. If true, the kernel flushes all pages within the range.

lock_value

[in scalar] One or more forms of access **not** permitted for the specified data. Valid values are:

VM_PROT_NO_CHANGE

Do not change the protection of any pages.

VM_PROT_NONE

Prohibits no access (that is, all forms of access are permitted).

VM_PROT_ALL

Prohibits all forms of access.

Other valid values are obtained by ORing together the following values:

VM_PROT_READ

Prohibits read access.

VM_PROT_WRITE

Prohibits write access.

VM_PROT_EXECUTE

Prohibits execute access.

reply_to

[in reply receive (to be converted to send) right] The response port to be used by the kernel on a call to **memory_object_lock_completed**, or MACH_PORT_NULL if no response is required.

NOTES

The **memory_object_lock_request** call affects only data that is cached at the time of the call. Access restrictions cannot be applied to pages for which data has not been provided.

When a running thread requires an access that is currently prohibited, the kernel issues a **memory_object_data_unlock** call specifying the access required. The memory manager can then use **memory_object_lock_request** to relax its access restrictions on the data.

RETURN VALUE

Since this function is asynchronous, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_data_supply**, **memory_object_data_unlock**, **memory_object_lock_completed**.

memory_object_representative

Function — Associates a memory object representative with an already initialized memory object

SYNOPSIS

kern_return_t **memory_object_representative**

(mach_port_t	*host,*
mach_port_t	*mem_cache_control,*
mach_port_t	*representative*,
mach_port_t	*abs_rep,*
vm_prot_t	*rep_max_prot*,
security_id_t	*security_id*);

DESCRIPTION

The **memory_object_representative** function informs the kernel of the identity of the abstract memory object (port) underlying a given memory object representative as well as the maximum protection implied by the representative. This call is the memory manager's reply to a previous **memory_object_notify** call. This call is used (as opposed to **memory_object_establish**) when the abstract memory object underlying the representative has been initialized.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host upon which the object is being mapped. The name port is provided by the kernel in the **memory_object_notify** call.

mem_cache_control
: [in memory-cache-control send right (moved instead of copied)] The control port to the memory cache object created for the representative being mapped. This port is provided by the kernel in the **memory_object_notify** call.

representative
: [in memory-object-representative receive (to be converted to send) right] The memory object representative port from the corresponding **memory_object_notify** message.

abs_rep
: [in abstract-memory-object receive (to be converted to send) right] The underlying abstract memory object. All subsequent messages from the kernel will come over this port.

rep_max_prot
: [in scalar] The maximum protections that can be specified by any task mapping this object via the memory object representative port over which the initial **memory_object_notify** message arrived.

security_id
: [in scalar] The security ID that tasks attempting to map this representative must have. A zero value indicates all identities.

NOTES

When a **memory_object_notify** message is sent by the kernel, it indicates that the kernel did not know the identity of a supplied memory object representative. Assuming that the representative represents a new memory object, the kernel creates a memory cache control port for it. If the object had not been initialized, the memory manager would respond with **memory_object_establish**, thereby initializing the memory object. The manager would keep the memory cache control port as its way to manipulate the cache object.

Since the kernel does not know the identity of the representative, the kernel must be told. In the case where the underlying memory object is initialized, the **memory_object_representative** call is used (as opposed to **memory_object_establish**). **memory_object_representative** differs from **memory_object_establish** in the following ways:

- **memory_object_representative** does not include the object attributes since the object is already initialized.
- **memory_object_representative** moves (gives away) the memory cache control port send right instead of copying it. This is an optimization, since the kernel will destroy this port when it receives this message. (The kernel does this since no new memory cache object is created in this case and the control port supplied in the notify message does not reference the already existing memory cache object.)
- Representative initialization may race with memory object termination. When the manager responds with **memory_object_representative**, the manager believes the memory object to be initialized. The object, though, may have been terminated but the manager has yet to receive the **memory_object_terminate** message. The use of **memory_object_representative** in this situation causes the kernel to discard the message (since the underlying object is not known) and to resend the **memory_object_notify** message, thereby establishing a new cache control port. When the manager sees the terminate message, it then re-initializes the memory object with **memory_object_establish**.

Unless the manager is absolutely sure that the memory object implied by the **memory_object_notify** message is un-initialized (because it has never been initialized or the **memory_object_terminate** message has been received), the manager's response to **memory_object_notify** should be **memory_object_representative**.

If the kernel was not expecting this message (the kernel was not in the process of initializing the representative), the kernel simply discards it. This is indistinguishable from the case where the kernel accepts this message and then immediately forgets the representative-to-abstract-object association.

RETURN VALUE

Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_notify**, **memory_object_establish**.

memory_object_revoke

Function — Revoke access to a particular user of a memory object

SYNOPSIS

kern_return_t **memory_object_revoke**
(mach_port_t *mem_cache_control,*
mach_port_t *mem_obj_rep*);

DESCRIPTION

The **memory_object_revoke** function causes any task that has gained access to the specified memory object through the specified representative to lose access. The object will not be unmapped from the tasks' address spaces, but all future accesses to the memory object will fail.

PARAMETERS

mem_cache_control
: [in memory-cache-control send right] The control port to the memory cache object for the representative in question. This port is provided by the kernel in the **memory_object_notify** call.

mem_obj_rep
: [in memory-object-representative receive (to be converted to send) right] The memory object representative port used by those tasks whose access is to be revoked.

RETURN VALUE

Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_notify**, **memory_object_establish**, **device_close**.

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.

mach_thread_self

System Trap — Returns the thread self port

SYNOPSIS

mach_port_t **mach_thread_self**
();

DESCRIPTION

The **mach_thread_self** function returns send rights to the thread's self special port.

PARAMETERS

None

RETURN VALUE

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

RELATED INFORMATION

Functions: **thread_set_special_port**.

thread_continue_state

System Trap — Continue thread execution with specified state

SYNOPSIS

kern_return_t **thread_continue_state**

(mach_port_t	*target_thread,*
thread_state_flavor_t	*flavor,*
thread_state_t	*new_state,*
mach_msg_type_number_t	*new_stateCnt,*
mach_msg_size_t	*rcv_size,*
mach_port_t	*rcv_name,*
mach_msg_header_t*	*rcv_msg*);

DESCRIPTION

The **thread_continue_state** trap causes the target thread to continue with the specified state from the point at which it took its last exception. The format of the state to be set is machine specific; it is defined in **<mach/thread_status.h>**. If the target thread is not waiting for the completion of exception processing, the effect of this call on the target thread is undefined.

Unless *rcv_name* is MACH_PORT_NULL, this trap then receives a message on the port or port set specified by *rcv_name*. The message is received in the *rcv_message* buffer; messages larger than *rcv_size* are not received. This feature is provided so as to be able to restart a thread and receive a subsequent exception message in one kernel call.

PARAMETERS

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

flavor
[in scalar] The type of state to set; valid values correspond to the supported machine architecture features.

new_state
[pointer to in structure] State information with which the target thread is to be continued. The maximum size of this array is THREAD_STATE_MAX.

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

rcv_size
: [in scalar] The maximum size, in bytes, of the message buffer. If *rcv_name* is MACH_PORT_NULL, this parameter is ignored.

rcv_name
: [in random right] If receiving a message after the target thread has been continued, *rcv_name* specified the port or port set to which the message is sent. Otherwise, MACH_PORT_NULL should be supplied.

rcv_msg
: [out structure] A message buffer in which this trap receives a message. The buffer must be naturally aligned.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_exception_continue**, **thread_notify**.

thread_create

Function — Creates a thread within a task

SYNOPSIS

```
kern_return_t thread_create
                (mach_port_t                    parent_task,
                 mach_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 name port. A send right for the thread's name 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. Its scheduling policy and processor set are inherited from its parent task.

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

Function — Optimized creation of a running thread

SYNOPSIS

kern_return_t **thread_create_running**

(mach_port_t	*parent_task,*
thread_state_flavor_t	*flavor,*
thread_state_t	*new_state,*
mach_msg_type_number_t	*new_stateCnt,*
mach_port_t*	*child_thread*);

DESCRIPTION

The **thread_create_running** function creates a new thread within *parent_task* with the specified state. The format of the state to be set is machine specific; it is defined in **<mach/thread_status.h>**.

The new thread is not suspended and holds a send right for its thread kernel port. A send right for the thread's name 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, and its scheduling policy and processor set are inherited from its parent task.

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 set; valid values correspond to the supported machine architecture features.

new_state
: [pointer to in structure] State information for the specified thread. The maximum size of this array is THREAD_STATE_MAX.

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

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

System Trap — Cancel thread scheduling depression

SYNOPSIS

kern_return_t **thread_depress_abort**
(mach_port_t *target_thread*);

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_switch**.

thread_exception_continue

Function — Indicates completion of thread exception processing

SYNOPSIS

```
kern_return_t thread_exception_continue
        (mach_port_t                    target_thread);
```

thread_exception_continue_state

Expanded form

```
kern_return_t thread_exception_continue_state
        (mach_port_t                    target_thread,
        thread_state_flavor_t           flavor,
        thread_state_t                  new_state,
        mach_msg_type_number_t          new_stateCnt);
```

DESCRIPTION

The **thread_exception_continue** function causes the target thread to continue from the point at which it took its last exception. If the target thread is not waiting for the completion of exception processing, the effect of this call on the target thread is undefined.

In the expanded form, one can also specify the state with which the thread is to continue. The format of the state to be set is machine specific; it is defined in **<mach/thread_status.h>**.

PARAMETERS

target_thread
: [in thread send right] The thread to continue.

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

new_state
: [pointer to in structure] State information with which the target thread is to be continued. The maximum size of this array is THREAD_STATE_MAX.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_notify**, **thread_set_state**, **catch_exception_raise**.

thread_get_exception_ports

Function — Returns a send right to an exception port

SYNOPSIS

```
kern_return_t thread_get_exception_ports
        (mach_port_t                     target_thread,
        exception_mask_t                 exception_mask,
        exception_mask_array_t           masks,
        mach_msg_type_number_t*          masksCnt,
        mach_port_array_t                old_ports,
        exception_behavior_array_t       old_behaviors,
        exception_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

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

exception_mask
: [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.

EXC_MASK_NOTIFY
Notify (**thread_notify**) 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.

masksCnt
[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_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_status.h>**.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_get_special_port

Function — Returns a send right to a special port

SYNOPSIS

kern_return_t **thread_get_special_port**
(mach_port_t *target_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**
(mach_port_t *target_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*.

PARAMETERS

target_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

Function — Returns the execution state for a thread

SYNOPSIS

kern_return_t **thread_get_state**

(mach_port_t	*target_thread,*
thread_state_flavor_t	*flavor,*
thread_state_t	*old_state*,
mach_msg_type_number_t*	*old_stateCnt*);

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 machine architectures.

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

old_stateCnt
[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**.

thread_info

Function — Returns information about a thread

SYNOPSIS

```
kern_return_t thread_info
        (mach_port_t                     target_thread,
         thread_flavor_t                        flavor,
         thread_info_t                 thread_info_out,
         mach_msg_type_number_t*    thread_info_outCnt);
```

DESCRIPTION

The **thread_info** function returns an information structure of type *flavor*, described in **<mach/thread_info.h>**.

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
[out structure] Information about the specified thread.

thread_info_outCnt
[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**.

thread_notify

Function — Provide a notification when a thread is at a clean point

SYNOPSIS

kern_return_t **thread_notify**
(mach_port_t *target_thread*);

DESCRIPTION

The **thread_notify** function causes *target_thread* to take a NOTIFY exception when it is next at a "clean point". A thread is at a clean point if it is executing in user space, or suspended waiting to exit kernel space.

PARAMETERS

target_thread
[in thread send right] The thread to test.

NOTES

The **thread_notify** function is used to resolve races between thread execution and server execution. It allows a server to wait for the target thread to complete any kernel processing before modifying the thread's state, thereby allowing the server to control the behavior of a thread:

thread_notify (*target_thread*)
wait for the notification
thread_exception_continue_state (*target_thread*)

If the thread is suspended when this call is made, it will generate the exception; but when it gets the exception reply, it will not run until it is resumed.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_resume

Function — Resumes a thread

SYNOPSIS

kern_return_t **thread_resume**
(mach_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

Function — Perform periodic PC sampling for a thread

SYNOPSIS

kern_return_t **thread_sample**
(mach_port_t *sample_thread*,
mach_port_t *reply_port*);

DESCRIPTION

The **thread_sample** function causes the program counter (PC) of the specified *sample_thread* to be sampled periodically. The set of PC sample values obtained are saved in buffers that 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**.

thread_set_exception_ports

Function — Sets exception ports for a thread

SYNOPSIS

kern_return_t **thread_set_exception_ports**
(mach_port_t *thread,*
exception_mask_t *exception_mask,*
mach_port_t *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_mask
[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.

EXC_MASK_NOTIFY
Notify (**thread_notify**) 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_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.

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

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_set_special_port

Function — Sets a special port for a thread

SYNOPSIS

```
kern_return_t thread_set_special_port
        (mach_port_t              target_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
        (mach_port_t              target_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

target_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 (self) port. 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**.

thread_set_state

Function — Sets the execution state for a thread

SYNOPSIS

kern_return_t **thread_set_state**

(mach_port_t	*target_thread,*
thread_state_flavor_t	*flavor,*
thread_state_t	*new_state*,
mach_msg_type_number_t	*new_stateCnt*);

DESCRIPTION

The **thread_set_state** function sets the execution state (for example, the machine registers) for *target_thread*. The value of *flavor* specifies the type of state to set. The format of the state to be 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 the supported machine architecture features.

new_state
: [pointer to in structure] State information for the specified thread. The maximum size of this array is THREAD_STATE_MAX.

new_stateCnt
: [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

Function — Suspends a thread

SYNOPSIS

kern_return_t **thread_suspend**
(mach_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 will continue to execute until it reaches the system return code. 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.

NOTES

The **thread_suspend** function can be used in conjunction with **thread_set_state** and **thread_resume** to control the behavior of a thread, provided that there is no interference from other controlling threads:

thread_suspend (*target_thread*)
thread_set_state (*target_thread*)
thread_resume (*target_thread*)

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_swap_exception_ports

Function — Swaps exception ports for a thread

SYNOPSIS

kern_return_t **thread_swap_exception_ports**

(mach_port_t	*thread,*
exception_mask_t	*exception_mask,*
mach_port_t	*port*,
exception_behavior_t	*behavior*,
thread_state_flavor_t	*flavor*,
exception_mask_array_t	*masks*,
mach_msg_type_number_t*	*masksCnt*,
mach_port_array_t	*old_ports*,
exception_behavior_array_t	*old_behaviors*,
exception_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_mask
: [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.

EXC_MASK_NOTIFY
: Notify (**thread_notify**) 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_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.

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

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.

masksCnt
: [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_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_status.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.

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_set_exception_ports**, **catch_exception_raise**.

thread_switch

System Trap — Cause context switch with options

SYNOPSIS

kern_return_t **thread_switch**
(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. The scheduler's selection logic is invoked to select the next thread to run; the current thread may continue running if there is no other appropriate thread to run. The *option* argument specifies additional semantics.

PARAMETERS

option
[in scalar] One of the options specified.

SWITCH_OPTION_NONE
The *time* argument is ignored. The current thread will yield to any runnable thread at the same or higher priority.

SWITCH_OPTION_WAIT
The current thread is blocked for the specified *time*. This wait cannot be canceled by **thread_resume**; only **thread_terminate** can abort this wait. If *time* is 0, the function is a NOP.

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 (because the processor set has nothing else to run), or when **thread_depress_abort** is applied to the current thread. Changing the thread's scheduling attributes (via **thread_policy** or **thread_set_policy**) will not affect this depression. If *time* is 0, no timeout is associated with the depression

SWITCH_OPTION_IDLE
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 or when **thread_depress_abort** is applied to the current thread.

KERN_ABORTED is returned if the depression is cancelled via **thread_depress_abort**. Changing the thread's scheduling attributes (via **thread_policy** or **thread_set_policy**) or applying **thread_resume** 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**.

time

[in scalar] Time duration in milliseconds 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 the thread to be transmitted to the kernel. 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. SWITCH_OPTION_IDLE effectively allows an idle system to busy wait in user space. The time spent executing in user space under scheduling depression is recorded separately from user and system time. It can be obtained as the *user_idle_time* value from the THREAD_BASIC_INFO flavor of thread_info for a specific thread, or as the *user_idle_time* value from the TASK_BASIC_INFO or TASK_THREAD_TIMES_INFO flavors of **task_info**. SWITCH_OPTION_DEPRESS is recommended when the time cannot be reasonable estimated.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_depress_abort**.

thread_terminate

Function — Destroys a thread

SYNOPSIS

kern_return_t **thread_terminate**
(mach_port_t *target_thread*);

DESCRIPTION

The **thread_terminate** function destroys *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_notify**, **thread_resume**, **thread_suspend**.

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.

mach_task_self

System Trap — Returns the task self port

SYNOPSIS

mach_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

Function — Creates a task

SYNOPSIS

```
kern_return_t task_create
                (mach_port_t                    ledger,
                 mach_port_t                    parent,
                 boolean_t                      inherit_memory,
                 mach_port_t*                   child);
```

DESCRIPTION

The **task_create** function creates a new task from *parent* and returns the name of the new task in *child*. 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, processor set, and scheduling policy.

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 a Name Server). 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.
- [ledger send right] The port naming the ledger from which the task draws its resources.

The child task also inherits the following ports:

- [exception send rights] Ports to which exception messages are sent.

PARAMETERS

ledger

[in ledger-name send right] Resource ledger from which the task will draw its resources.

parent

[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

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

RETURN VALUE

KERN_INVALID_TASK

parent does not name a task.

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**.

task_get_exception_ports

Function — Returns send rights to exception ports

SYNOPSIS

kern_return_t **task_get_exception_ports**

(mach_port_t	*target_task,*
exception_mask_t	*exception_mask*,
exception_mask_array_t	*masks*,
mach_msg_type_number_t*	*masksCnt*,
mach_port_array_t	*old_ports*,
exception_behavior_array_t	*old_behaviors*,
exception_flavor_array_t	*old_flavors*);

DESCRIPTION

The **task_get_exception_ports** function returns send rights for a specified set of exception ports belonging to *target_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

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

exception_mask
: [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.

EXC_MASK_NOTIFY
Notify (**thread_notify**) 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.

masksCnt
[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_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 *exception_flavor_t*] The type of state to be sent with the exception message. These types are defined in **<mach/thread_status.h>**.

RETURN VALUE

Only generic errors apply.

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**.

task_get_special_port

Function — Returns a send right to a special port

SYNOPSIS

kern_return_t **task_get_special_port**

(mach_port_t	*target_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**

(mach_port_t	*target_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**

(mach_port_t	*target_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**

(mach_port_t	*target_task,*
mach_port_t*	*special_port)*

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

task_get_ledger_port

Macro form

kern_return_t **task_get_ledger_port**

(mach_port_t	*target_task,*
mach_port_t*	*special_port)*

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

DESCRIPTION

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

target_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**.

TASK_LEDGER_PORT
: [ledger-name send right] The port naming the source from which this task draws its resources.

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**.

task_info

Function — Returns information about a task

SYNOPSIS

kern_return_t **task_info**

(mach_port_t	*target_task,*
task_flavor_t	*flavor,*
task_info_t	*task_info_out*,
mach_msg_type_number_t*	*task_info_outCnt*);

DESCRIPTION

The **task_info** function returns an information structure of type *flavor*, described in **<mach/task_info.h>**.

PARAMETERS

target_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
[out structure] Information about the specified task.

task_info_outCnt
[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**.

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

task_resume

Function — Resume a task

SYNOPSIS

kern_return_t **task_resume**
(mach_port_t *target_task*);

DESCRIPTION

The **task_resume** function decrements the suspend count for *target_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

target_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

Function — Perform periodic PC sampling for a task

SYNOPSIS

kern_return_t **task_sample**
(mach_port_t *sample_task*,
mach_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. 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**.

task_set_exception_ports

Function — Sets exception ports for a task

SYNOPSIS

kern_return_t **task_set_exception_ports**

(mach_port_t	*target_task,*
exception_mask_t	*exception_mask,*
mach_port_t	*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 *target_task*. A task exception port is used when a thread specific exception port returns a non-success reply.

PARAMETERS

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

exception_mask
: [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.

EXC_MASK_NOTIFY
Notify (**thread_notify**) 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_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.

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

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**.

task_set_special_port

Function — Sets a special port for a task

SYNOPSIS

```
kern_return_t task_set_special_port
        (mach_port_t               target_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
        (mach_port_t               target_task,
         mach_port_t               special_port)
```

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

task_set_kernel_port

Macro form

```
kern_return_t task_set_kernel_port
        (mach_port_t               target_task,
         mach_port_t               special_port)
```

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

task_set_host_name_port

Macro form

```
kern_return_t task_set_host_name_port
        (mach_port_t               target_task,
         mach_port_t               special_port)
```

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

task_set_ledger_port

Macro form

```
kern_return_t task_set_ledger_port
        (mach_port_t               target_task,
         mach_port_t               special_port)
```

⇒ **task_set_special_port** (*target_task*, TASK_LEDGER_PORT, *special_port*)

DESCRIPTION

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

PARAMETERS

target_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.

TASK_LEDGER_PORT
[ledger-name send right] The resource ledger from which the task draws its resources. Setting this special port does not change the ledger from which the task draws its resources; this simply changes the value returned as the ledger 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**.

task_suspend

Function — Suspends a task

SYNOPSIS

kern_return_t **task_suspend**
(mach_port_t *target_task*);

DESCRIPTION

The **task_suspend** function increments the suspend count for *target_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

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

NOTES

The suspend count for existing threads within a task is incremented each time **task_suspend** is called and decremented each time **task_resume** is called The suspend count for newly created threads is inherited from the parent task. A thread is resumed when its suspend count reaches 0, irrespective of the suspend count of the parent task.

To resume a suspended task and its threads, use **task_resume**. If the suspend count is greater than 1, **task_resume** must be called 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

Function — Swaps exception ports for a task

SYNOPSIS

```
kern_return_t task_swap_exception_ports
                (mach_port_t                          target_task,
                 exception_mask_t                     exception_mask,
                 mach_port_t                          port,
                 exception_behavior_t                 behavior,
                 thread_state_flavor_t                flavor,
                 exception_mask_array_t               masks,
                 mach_msg_type_number_t*              masksCnt,
                 mach_port_array_t                    old_ports,
                 exception_behavior_array_t           old_behaviors,
                 exception_flavor_array_t             old_flavors);
```

DESCRIPTION

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

PARAMETERS

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

exception_mask
[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.

EXC_MASK_NOTIFY
Notify (**thread_notify**) 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_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.

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

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.

masksCnt
[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_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_status.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.

RETURN VALUE

Only generic errors apply.

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**.

task_terminate

Function — Destroys a task

SYNOPSIS

kern_return_t **task_terminate**
(mach_port_t *target_task*);

DESCRIPTION

The **task_terminate** function destroys *target_task* and all its threads, if any.

PARAMETERS

target_task
[in task send right] 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

Function — Returns a list of the threads within a task

SYNOPSIS

```
kern_return_t task_threads
        (mach_port_t                    target_task,
         mach_port_array_t              threads,
         mach_msg_type_number_t*        threadsCnt);
```

DESCRIPTION

The **task_threads** function returns a send right to the kernel port for each thread within the target task. The rights are returned in the array *threads*.

PARAMETERS

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

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

threadsCnt
: [in/out scalar] On input, the size of *threads*; on output, the returned count of threads in *threads*. The maximum size of *threads* is TASK_THREADS_MAX.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

CHAPTER 8 Host, Clock and Alarm 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 parameters.

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. The control port for a host is provided to the bootstrap task for that host. Only the name port operations are discussed in this chapter.

Associated with each host is a (small) set of clocks used to keep track of time and to generate alarms (via alarm objects) at requested times. Each kernel clock object has both a name and a control port. Only the name port operations are discussed in this chapter.

alarm_create

Function — Create an alarm object

SYNOPSIS

kern_return_t **alarm_create**
(mach_port_t *target_ledger*,
mach_port_t *clock*,
mach_port_t *alarm_reply_port*,
mach_port_t* *alarm*);

DESCRIPTION

The **alarm_create** function creates an alarm object that can be set to generate alarm notification messages at specified future times.

PARAMETERS

target_ledger
[in ledger-name send right] Resource ledger from which the alarm object will draw its resources.

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

alarm_reply_port
[in destination or notification port send right] Target port for all alarm notification messages.

alarm
[out alarm send right] A port naming the alarm object.

NOTES

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

RETURN VALUE

KERN_INVALID_CLOCK
The *clock* supplied does not name a clock.

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**, **alarm_set**, **alarm_cancel**, **alarm_terminate**.

alarm_cancel

Function — Cancel a pending or repeating alarm

SYNOPSIS

kern_return_t **alarm_cancel**
(mach_port_t *alarm*);

DESCRIPTION

The **alarm_cancel** function requests that a pending or repeating alarm be canceled.

PARAMETERS

alarm
[in alarm send right] A port naming the alarm object.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **alarm_create**, **alarm_set**, **alarm_terminate**, **alarm_reply**.

alarm_set

Function — Arm an alarm object

SYNOPSIS

kern_return_t **alarm_set**

(mach_port_t	*alarm*,
alarm_type_t	*alarm_type*,
tvalspec_t	*alarm_time*,
tvalspec_t	*repeat_time*);

DESCRIPTION

The **alarm_set** function arms an alarm object, which will result in future (or a series of periodic future) alarm notification messages. The format of alarm notification messages is specified by **alarm_reply**. Setting an alarm cancels the previous setting of the same alarm object.

PARAMETERS

alarm
[in alarm send right] A port naming the alarm object.

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 first alarm is to be sent.

repeat_time
[in structure] How much time should pass between successive alarm messages. A zero value indicates that this is a non-repeating (one-shot) alarm.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **alarm_create**, **alarm_cancel**, **alarm_terminate**, **alarm_reply**.

Data Structures: **tvalspec**.

alarm_terminate

Function — Destroy an alarm object

SYNOPSIS

kern_return_t **alarm_terminate**
(mach_port_t *alarm*);

DESCRIPTION

The **alarm_terminate** function destroys an alarm object, cancelling any pending or repeating alarms.

PARAMETERS

alarm
[in alarm send right] A port naming the alarm object.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **alarm_create**, **alarm_set**, **alarm_cancel**, **alarm_reply**.

clock_get_attributes

Function — Returns attributes of a clock

SYNOPSIS

kern_return_t **clock_get_attributes**

(mach_port_t	*clock,*
clock_flavor_t	*flavor,*
clock_attr_t	*clock_attr*,
mach_msg_type_number_t*	*clock_attrCnt*);

DESCRIPTION

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

PARAMETERS

clock
[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.

clock_attr
[out scalar] The returned attribute.

clock_attrCnt
[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**, **alarm_set**, **clock_set_attributes**.

clock_get_time

Function — Returns the current time

SYNOPSIS

```
kern_return_t clock_get_time
        (mach_port_t        clock,
         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
: [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**, **alarm_set**, **clock_set_time, get_time_stamp**.

Data Structures: **tvalspec**.

clock_map_time

Function — Returns a memory object that maps a clock

SYNOPSIS

```
kern_return_t clock_map_time
                (mach_port_t          clock,
                 mach_port_t*         pager);
```

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. The returned right is suitable as an argument for **vm_map**.

PARAMETERS

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

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

NOTES

Not all clocks provide this service, but the REALTIME clock must. The format of the REALTIME clock's map is **mapped_tvalspec**.

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**, **alarm_create**.

Data Structures: **mapped_tvalspec**.

clock_sleep

System Trap — Sleep until a given time

SYNOPSIS

```
kern_return_t clock_sleep
        (mach_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.

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 terminated via use of **clock_set_time**.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **alarm_set**, **clock_set_time**.

Data Structures: **tvalspec**.

get_time_stamp

System Trap— Quickly return high accuracy time

SYNOPSIS

kern_return_t **get_time_stamp**
(tvalspec_t* *cur_time*);

DESCRIPTION

The **get_time_stamp** function returns the current time kept by the real-time clock, to high accuracy. The time is suitable for use as an event time-stamp.

PARAMETERS

cur_time
[out structure] Current time

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **clock_get_time**.

Data Structures: **tvalspec**.

host_get_clock_service

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

SYNOPSIS

kern_return_t **host_get_clock_service**

(mach_port_t	*host,*
clock_id_t	*clock_id,*
mach_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.

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**, **alarm_create**, **host_get_clock_control**.

host_info

Function — Returns information about a host

SYNOPSIS

kern_return_t **host_info**

(mach_port_t	*host,*
host_flavor_t	*flavor,*
host_info_t	*host_info_out*,
mach_msg_type_number_t*	*host_info_outCnt*);

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. The name port is the same as that returned by **mach_host_self**.

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_info_out
: [out structure] Information about the specified host.

host_info_outCnt
: [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**, **mach_host_self**.

Data Structures: **host_basic_info, host_sched_info**.

host_kernel_version

Function — Returns kernel version information for a host

SYNOPSIS

kern_return_t **host_kernel_version**
(mach_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. The name port is the same as that returned by **mach_host_self**.

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**, **mach_host_self**.

mach_host_self

System Trap — Returns the host self port

SYNOPSIS

mach_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 task's host self 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 set interfaces. Note that processor sets have two ports that name them: a name port that allows information to be requested about them, and a control port that allows full access.

processor_set_default

Function — Returns the default processor set

SYNOPSIS

kern_return_t **processor_set_default**
(mach_port_t *host,*
mach_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. The host name port is that returned by **mach_host_self**.

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**, **mach_host_self**.

processor_set_info

Function — Returns information about a processor set.

SYNOPSIS

kern_return_t **processor_set_info**

(mach_port_t	*processor_set_name,*
processor_set_flavor_t	*flavor,*
mach_port_t*	*host,*
processor_set_info_t	*pset_info*,
mach_msg_type_number_t*	*pset_infoCnt*);

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. This port can be obtained from **host_processor_set_priv**.

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.

pset_info
[out structure] Information about the processor set.

pset_infoCnt
[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). Since newly created tasks and threads inherit their scheduling policies from their parents, a task or thread inherits only the processor set default policy when its policy is disabled. However, a processor set maintains bases and limits for all defined scheduling policies, so all of the scheduling limit structures and scheduling base structures return valid values.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**:** **host_processor_set_priv**, **processor_set_statistics**, **processor_set_default**, **processor_set_policy_control**.

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

task_get_assignment

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

SYNOPSIS

kern_return_t **task_get_assignment**
(mach_port_t *task,*
mach_port_t* *pset*);

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.

pset
[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**: processor_set_info**, **mach_host_self**.

task_policy

Function — Sets the default scheduling attributes for a task

SYNOPSIS

kern_return_t **task_policy**

(mach_port_t	*task,*
policy_t	*policy,*
policy_base_t	*base*,
mach_msg_type_number_t	*baseCnt*,
boolean_t	*set_limit*,
boolean_t	*change*);

DESCRIPTION

The **task_policy** function sets the default scheduling attributes for *task*. These attributes are used when creating new tasks and threads. The policy must be enabled for the task's assigned processor set, and the base and limit may not exceed the policy-specific limit for the task's assigned processor set.

Changing the default attributes for a task does not affect the attributes of the contained threads unless *change* 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**.

baseCnt

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

set_limit

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

change
[in boolean] 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: **mach_task_self**, **thread_policy**, **thread_set_policy**, **task_set_policy**, **processor_set_info**, **processor_set_policy_control**.

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

thread_get_assignment

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

SYNOPSIS

```
kern_return_t thread_get_assignment
        (mach_port_t        thread,
         mach_port_t*       pset);
```

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.

pset
: [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**: mach_thread_self**, **processor_set_info**.

thread_policy

Function — Sets the scheduling policy to apply to a thread

SYNOPSIS

```
kern_return_t thread_policy
        (mach_port_t                 thread,
         policy_t                    policy,
         policy_base_t               base,
         mach_msg_type_number_t      baseCnt,
         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, and the base and limit may not exceed the policy-specific limit 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**.

baseCnt
: [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**.

CHAPTER 10 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.

device_close

Function — Terminate a connection to a device.

SYNOPSIS

kern_return_t **device_close**
(mach_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 and other indirect device references 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**, **memory_object_revoke**.

device_get_status

Function — Return the current device status

SYNOPSIS

```
kern_return_t device_get_status
        (mach_port_t                  device,
         dev_flavor_t                 flavor,
         dev_status_t                 status,
         mach_msg_type_number_t*      statusCnt);
```

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.

PARAMETERS

device
: [in device send right] A device port to the device to be interrogated.

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

status
: [out array of natural-sized units] The returned device status.

statusCnt
: [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_INVALID_OPERATION
: The *flavor* requested is inconsistent with the device type or opening mode.

D_NO_SUCH_DEVICE
: The device is not open, or the device opening is restricted to a single identity other than that of the caller.

D_OUT_OF_BAND
Out-of-band condition occurred on device (such as typing control-C)

RELATED INFORMATION

Functions: **device_set_status**.

device_map

Function — Establish a memory manager representing a device

SYNOPSIS

```
kern_return_t device_map
        (mach_port_t          device,
         vm_prot_t            prot,
         vm_size_t*           size,
         mach_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
: [out scalar] The size of the device memory object. All mappings of the same device receive the same size.

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.

Only one memory object representative can be in effect for any given device opening. A call to **device_map** that specifies the same device as a previous call to **device_map** destroys the existing representative before creating the new one.

CAUTIONS

Memory provided by **device_map** may only be mapped by tasks on the same host as the device is attached. The representative port supplied is actually a dead right, so attempts to **vm_map** the representative on any other node or to use the representative right in any other way will fail.

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_NO_SUCH_DEVICE
: The device is not open, its opening mode is incompatible with the request, or the device is restricted to a single identity other than that of the caller.

D_READ_ONLY
: Data cannot be written to this device.

RELATED INFORMATION

Functions: **vm_map**.

device_read

Function — Read a sequence of bytes from a device object.

SYNOPSIS

```
kern_return_t device_read
                    (mach_port_t                                   device,
                    dev_mode_t                                     mode,
                    recnum_t                                       recnum,
                    io_buf_len_t                                   data_count,
                    kern_return_t*                                 device_code,
                    pointer_t*                                     data,
                    mach_msg_type_number_t*                        dataCnt);
```

device_read_inband

In-line Function form — Read data in-line

```
kern_return_t device_read_inband
                    (mach_port_t                                   device,
                    dev_mode_t                                     mode,
                    recnum_t                                       recnum,
                    io_buf_len_t                                   data_count,
                    kern_return_t*                                 device_code,
                    pointer_t                                      data,
                    mach_msg_type_number_t*                        dataCnt);
```

device_read_request

Asynchronous Function form — Asynchronously read data

```
kern_return_t device_read_request
                    (mach_port_t                                   device,
                    mach_port_t                                    reply_port,
                    dev_mode_t                                     mode,
                    recnum_t                                       recnum,
                    io_buf_len_t                                   data_count);
```

device_read_request_inband

Asynchronous In-line Function form — Asynchronously read data in-line

```
kern_return_t device_read_request_inband
                    (mach_port_t                                   device,
                    mach_port_t                                    reply_port,
                    dev_mode_t                                     mode,
                    recnum_t                                       recnum,
                    io_buf_len_t                                   data_count);
```

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	*dataCnt*);

ds_device_read_reply_inband

Asynchronous In-line Server Interface form — Receive the reply from an asynchronous in-line read

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	*dataCnt*);

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.

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 upon completion of an asynchronous request.

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; used for positioning of random access devices.

data_count
: [in scalar] Size of data transfer, in bytes.

return_code
: [out scalar] The return status code from the read request.

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
  : The device is not open for reading or the device opening is restricted to a single identity other than that of the caller.

  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
: [array of bytes: out pointer to dynamic array for out-of-line form, out array for inband form, in pointer to dynamic array for asynchronous out-of-line form, in pointer to array for asynchronous inband form] Returned data bytes.

dataCnt
: [out scalar, in for asynchronous form] Number of returned data bytes.

NOTES

The default kernel interface stub uses a message reception option to cause the data return by the out-of-line forms to be wired. Different behavior can be obtained by modifying the MIG definition file

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_write**.

device_set_filter

Function — Names an input filter for a device

SYNOPSIS

kern_return_t **device_set_filter**

(mach_port_t	*device,*
mach_port_t	*rcv_port,*
mach_msg_type_name_t	*rcv_portPoly,*
net_priority_t	*priority,*
filter_array_t	*filter,*
mach_msg_type_number_t	*filterCnt*);

DESCRIPTION

The **device_set_filter** function provides a means by which network data appearing at a device interface can be selected and routed to a port. This service is provided in support of network devices. A given opening of a device can have only one filter attached (but each opening can have one); specifying a null filter port cancels the previous filter request for the opening.

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 that 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

rcv_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.

rcv_portPoly
: [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.

filterCnt
: [in scalar] The size of the *filter* array (in 16-bit values).

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_INVALID_OPERATION
: No filter port was supplied.

D_NO_SUCH_DEVICE
: The device is not open for reading or the device opening is restricted to a single identity other than that of the caller.

RELATED INFORMATION

Functions: **receive_packet**.

device_set_status

Function — Sets device status.

SYNOPSIS

kern_return_t **device_set_status**

(mach_port_t	*device,*
dev_flavor_t	*flavor,*
dev_status_t	*status,*
mach_msg_type_number_t	*statusCnt*);

DESCRIPTION

The **device_set_status** function sets device status. The possible values of *flavor* as well as the corresponding meanings are device dependent.

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.

status
: [pointer to in array of natural-sized units] The status information to set.

statusCnt
: [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 flavor requested is inconsistent with the device type or opening mode.

D_IO_ERROR
: Hardware IO error

D_NO_SUCH_DEVICE
: The device is not open or the device opening is restricted to a single identity other than that of the caller.

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

Function — Write a sequence of bytes to a device object.

SYNOPSIS

```
kern_return_t device_write
        (mach_port_t                 device,
         dev_mode_t                  mode,
         recnum_t                    recnum,
         pointer_t                   data,
         mach_msg_type_number_t      dataCnt,
         boolean_t                   dataDealloc,
         kern_return_t*              device_code,
         io_buf_len_t*               bytes_written);
```

device_write_inband

In-line Function form — Write data in-line

```
kern_return_t device_write_inband
        (mach_port_t                 device,
         dev_mode_t                  mode,
         recnum_t                    recnum,
         pointer_t                   data,
         mach_msg_type_number_t      dataCnt,
         kern_return_t*              device_code,
         io_buf_len_t*               bytes_written);
```

device_write_request

Asynchronous Function form — Asynchronously write data

```
kern_return_t device_write_request
        (mach_port_t                 device,
         mach_port_t                 reply_port,
         dev_mode_t                  mode,
         recnum_t                    recnum,
         pointer_t                   data,
         mach_msg_type_number_t      dataCnt,
         boolean_t                   dataDealloc);
```

device_write_request_inband

Asynchronous In-line Function form — Asynchronously write data in-line

```
kern_return_t device_write_request_inband
        (mach_port_t                 device,
         mach_port_t                 reply_port,
         dev_mode_t                  mode,
         recnum_t                    recnum,
         pointer_t                   data,
         mach_msg_type_number_t      dataCnt);
```

ds_device_write_reply

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*);

ds_device_write_reply_inband

Asynchronous In-line Server Interface form — Receive the reply from an asynchronous in-line 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** function writes 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 written.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent upon completion of an asynchronous request.

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; used for positioning of random access devices.

data
: [pointer to in array of bytes] Data to be written, in bytes.

dataCnt
: [in scalar] Number of data bytes to be written.

dataDealloc
: [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 request.

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
: The device is not open for writing or the device opening is restricted to a single identity other than that of the caller.

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, in bytes.

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_read**.

CHAPTER 11 Service Interfaces

This chapter discusses the specifics of the service interfaces called by the kernel.

alarm_reply

Server Interface — Indicates that an alarm went off

SYNOPSIS

```
kern_return_t alarm_reply
        (mach_port_t        alarm_reply_port,
         kern_return_t      alarm_code,
         alarm_type_t       alarm_type,
         tvalspec_t         wake_time);
```

DESCRIPTION

An **alarm_reply** function is called as the result of a message from the kernel indicating that a previously requested alarm time (**alarm_set**) has arrived.

PARAMETERS

alarm_reply_port
: [in alarm (receive) right] The notify port named in the corresponding **alarm_create** call.

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

KERN_SUCCESS
: The alarm was delivered without problem.

KERN_ABORTED
: The alarm was canceled.

alarm_type
: [in scalar] The alarm type value supplied to the **alarm_set** call. Only the low order byte of this value is used by the kernel so the high order bytes 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**, **alarm_create**, **alarm_set**, **alarm_cancel**, **alarm_reply_server**, **clock_set_time**.

Data Structures: **tvalspec**.

catch_exception_raise

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

SYNOPSIS

kern_return_t **catch_exception_raise**

(mach_port_t	*exception_port,*
mach_port_t	*thread,*
mach_port_t	*task,*
exception_type_t	*exception,*
exception_data_t	*code*);

catch_exception_raise_state

Expanded form

kern_return_t **catch_exception_raise_state**

(mach_port_t	*exception_port,*
exception_type_t	*exception,*
exception_data_t	*code*,
thread_state_flavor_t	*flavor*,
thread_state_t	*state*,
mach_msg_type_number_t	*stateCnt*);

catch_exception_raise_state_identity

Expanded form

kern_return_t **catch_exception_raise_state_identity**

(mach_port_t	*exception_port,*
mach_port_t	*thread*,
mach_port_t	*task*,
exception_type_t	*exception,*
exception_data_t	*code*,
thread_state_flavor_t	*flavor*,
thread_state_t	*state*,
mach_msg_type_number_t	*stateCnt*);

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. *code* 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 *code* field.

EXC_EMULATION
Emulation instruction. Emulation support instruction encountered. Details in *code* field.

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

EXC_BREAKPOINT
Trace, breakpoint, etc. Details in *code* field.

EXC_SYSCALL
System call requested. Details in *code* field.

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

EXC_NOTIFY
Notify (**thread_notify**) exception.

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

flavor
[in scalar] The type of state included as selected when the exception port was set.

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

state_count
[in 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 to be continued (**thread_exception_continue**). 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 simply call **thread_exception_continue**. If some other action should be taken, the thread's state can be explicitly set (**thread_set_state**) or set via **thread_exception_continue_state**. Either way, the thread will not proceed from the point of exception until explicitly continued.

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 send only a single exception message, to the thread exception port if set, otherwise to the task exception port. If neither port is set, the thread is terminated.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **exc_server**, **thread_exception_continue**, **thread_continue_state**, **task_get_exception_ports**, **thread_get_exception_ports**, **thread_get_state**, **task_set_exception_ports,** **thread_set_exception_ports**, **thread_set_state**, **thread_terminate.**

do_mach_notify_dead_name

Server Interface — Handles the occurrence of a dead-name notification

SYNOPSIS

kern_return_t **do_mach_notify_dead_name**
(mach_port_t *notify,*
mach_port_name_t *name)*;

do_seqnos_mach_notify_dead_name

Sequence Number form

kern_return_t **do_seqnos_mach_notify_dead_name**
(mach_port_t *notify,*
mach_port_seqno_t *seqno*,
mach_port_name_t *name)*;

DESCRIPTION

A **do_mach_notify_dead_name** function is called by **notify_server** as the result of a kernel message indicating that the port name is now dead as the result of the associated receive right having died. In contrast, a port-deleted notification indicates that the port name is no longer usable (that is, it no longer names a valid right), typically as a result of the right so named being consumed or moved. *notify* is the port named via **mach_port_request_notification** or **mach_msg**.

PARAMETERS

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

seqno
[in scalar] The sequence number of this message relative to the notification port.

name
[in scalar] The dead name.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **notify_server**, **seqnos_notify_server**, **mach_msg**, **mach_port_request_notification**, **do_mach_notify_port_deleted**, **do_mach_notify_send_once.**

do_mach_notify_port_deleted

Server Interface — Handles the occurrence of a port-deleted notification

SYNOPSIS

kern_return_t **do_mach_notify_port_deleted**
(mach_port_t *notify,*
mach_port_name_t *name)*;

do_seqnos_mach_notify_port_deleted

Sequence Number form

kern_return_t **do_seqnos_mach_notify_port_deleted**
(mach_port_t *notify,*
mach_port_seqno_t *seqno*,
mach_port_name_t *name)*;

DESCRIPTION

A **do_mach_notify_port_deleted** function is called by **notify_server** as the result of a kernel message indicating that a port name is no longer usable (that is, it no longer names a valid right), typically as a result of the right so named being consumed or moved. In contrast, a dead-name notification indicates that the port name is now dead as the result of the associated receive right having died. *notify* is the port named via **mach_port_request_notification** or **mach_msg**.

PARAMETERS

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

seqno
: [in scalar] The sequence number of this message relative to the notification port.

name
: [in scalar] The invalid name.

NOTES

This interface is machine word length specific because of the port name parameter.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **notify_server**, **seqnos_notify_server**, **mach_msg**, **mach_port_request_notification,** **do_mach_notify_dead_name**, **do_mach_notify_send_once.**

do_mach_notify_send_once

Server Interface — Handles the occurrence of a send-once notification

SYNOPSIS

kern_return_t **do_mach_notify_send_once**
(mach_port_t *random)*;

do_seqnos_mach_notify_send_once

Sequence Number form

kern_return_t **do_seqnos_mach_notify_send_once**
(mach_port_t random,
mach_port_seqno_t *seqno)*;

DESCRIPTION

A **do_mach_notify_send_once** function is called by **notify_server** as the result of a kernel message indicating that a send-once right was in any way destroyed. *notify* is the port for which a send-once right was destroyed.

PARAMETERS

random
[in notify (receive) right] The port whose send-once right was destroyed.

seqno
[in scalar] The sequence number of this message relative to the port.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **notify_server**, **seqnos_notify_server**, **mach_msg**, **mach_port_request_notification**, **do_mach_notify_port_deleted**, **do_mach_notify_dead_name.**

receive_packet

Server Interface — Handles the arrival of a device filter packet

SYNOPSIS

kern_return_t **receive_packet**

(mach_port_t	*reply_right,*
net_header_t	*header*,
io_buf_len_t	*packetCnt*,
net_packet_t	*packet*);

DESCRIPTION

A **receive_packet** function is called as the result of a kernel message indicating that a packet meeting the criteria of a device's packet filter has been received.

PARAMETERS

reply_right
: [in filter (receive) right] The port to which the packet message was sent.

header
: [in array of char] The packet header, of size NET_HDW_HDR_MAX.

packetCnt
: [in scalar] The number of bytes in the packet.

packet
: [in array of char] The packet body, of size NET_RCV_MAX.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **device_set_filter**, **device_reply_server.**

receive_samples

Server Interface — Handles the occurrence of a program counter (PC) sampling message

SYNOPSIS

kern_return_t **receive_samples**

(mach_port_t	*sample_port,*
sample_count_t	*samples_lost*,
sample_array_t	*samples*,
mach_msg_type_number_t	*samplesCnt*);

DESCRIPTION

A **receive_samples** function is called by **sample_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.

samples_lost
[out scalar] The number of samples that were lost because both sample buffer messages were waiting to be received from *sample_port*.

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

sampleCnt
[in scalar] The number of values in *sample*. A zero value indicates that no more samples will be forthcoming.

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**, **sample_server**.

CHAPTER 12

External Memory Management Service Interfaces

This chapter discusses the interfaces of an external memory manager invoked by the kernel as part of the kernel's external memory management protocols.

memory_object_change_completed

Server Interface — Indicates completion of an attribute change call

SYNOPSIS

kern_return_t **memory_object_change_completed**

(mach_port_t	*reply_port,*
mach_port_t	*mem_cache_control*,
memory_object_flavor_t	*flavor*);

seqnos_memory_object_change_completed

Sequence Number form

kern_return_t **seqnos_memory_object_change_completed**

(mach_port_t	*reply_port,*
mach_port_seqno_t	*seqno*,
mach_port_t	*mem_cache_control*,
memory_object_flavor_t	*flavor*);

DESCRIPTION

A **memory_object_change_completed** function is called as the result of a kernel message confirming the kernel's action in response to a **memory_object_change_attributes** call from the memory manager.

When the kernel completes the requested changes, it calls **memory_object_change_completed** (asynchronously) using the port explicitly provided in the **memory_object_change_attributes** call.

PARAMETERS

reply_port
: [in reply (receive) right] The reply port supplied in the corresponding **memory_object_change_attributes** call.

seqno
: [in scalar] The sequence number of this message relative to the port named in the **memory_object_change_attributes** call.

mem_cache_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.

flavor
: [in scalar] The flavor of attributes changed by the corresponding **memory_object_change_attributes** call. Valid values are:

MEMORY_OBJECT_PERFORMANCE_INFO
Performance related attributes such as the cache indicator and the cluster size. *attributes* should specify a structure of type **memory_object_perf_info**.

MEMORY_OBJECT_BEHAVIOR_INFO
Behavior related attributes such as the copy strategy and sync invalidate flag. *attributes* should specify a structure of type **memory_object_behave_info**.

NOTES

If the attribute change causes the termination of the object, no **memory_object_change_completed** message is sent. This case can be detected by the appearance of the **memory_object_terminate** message.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

Functions: **memory_object_change_attributes**, **memory_object_server**.

memory_object_data_request

Server Interface — Requests data from a memory object

SYNOPSIS

kern_return_t **memory_object_data_request**

(mach_port_t	*abs_mem_port,*
mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
vm_size_t	*length,*
vm_prot_t	*desired_access*);

seqnos_memory_object_data_request

Sequence Number form

kern_return_t **seqnos_memory_object_data_request**

(mach_port_t	*abs_mem_port,*
mach_port_seqno_t	*seqno*,
mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
vm_size_t	*length,*
vm_prot_t	*desired_access*);

DESCRIPTION

A **memory_object_data_request** function is called as the result of a kernel message requesting data from the specified memory object, for at least the access specified.

The kernel issues this call after a cache miss (that is, a page fault for which the kernel does not have the data). The kernel requests only amounts of data that are multiples of the page size included in the **memory_object_notify** call.

The memory manager is expected to use **memory_object_data_supply** to return at least the specified data, with as much access as it can allow. If the memory manager cannot provide the data (for example, because of a hardware error), it can use the **memory_object_data_error** call. The memory manager can also use **memory_object_data_unavailable** to tell the kernel to supply zero-filled memory for the region.

PARAMETERS

abs_mem_port

[in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

seqno
: [in scalar] The sequence number of this message relative to the abstract memory object port.

mem_cache_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. The offset must be on a page boundary.

length
: [in scalar] The number of bytes requested, starting at *offset*. The number converts to an integral number of virtual pages.

desired_access
: [in scalar] The memory access modes to be allowed for the cached data. Possible values are obtained by OR'ing together the following values:

  VM_PROT_READ
  : Allows read access.

  VM_PROT_WRITE
  : Allows write access.

  VM_PROT_EXECUTE
  : Allows execute access.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

Functions: **memory_object_data_error**, **memory_object_data_supply**, **memory_object_data_unavailable**, **memory_object_server**.

memory_object_data_return

Server Interface — Writes data back to a memory object

SYNOPSIS

kern_return_t **memory_object_data_return**

(mach_port_t	*abs_mem_port,*
mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
pointer_t	*data,*
mach_msg_type_number_t	*dataCnt*,
boolean_t	*dirty*,
boolean_t	*kernel_copy*);

seqnos_memory_object_data_return

Sequence Number form

kern_return_t **seqnos_memory_object_data_return**

(mach_port_t	*abs_mem_port,*
mach_port_seqno_t	*seqno*,
mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
pointer_t	*data,*
mach_msg_type_number_t	*dataCnt*,
boolean_t	*dirty*,
boolean_t	*kernel_copy*);

DESCRIPTION

A **memory_object_data_return** function is called as the result of a kernel message providing the memory manager with data that has been evicted from the physical memory cache. The kernel writes back only data that has been modified or is precious.

PARAMETERS

abs_mem_port

[in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

seqno

[in scalar] The sequence number of this message relative to the abstract memory object port.

mem_cache_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. The offset must be on a page boundary.

data
[in pointer to dynamic array of bytes] The data that has been evicted from the physical memory cache; must be on a page boundary.

dataCnt
[in scalar] The number of bytes to be written, starting at *offset*. The number converts to an integral number of memory object pages.

dirty
[in boolean] If TRUE, the pages returned have been modified.

kernel_copy
[in boolean] If TRUE, the kernel has kept a copy of the page.

NOTES

The kernel can flush clean (that is, un-modified) non-precious pages at its own discretion. As a result, the memory manager cannot rely on the kernel to keep a copy of its data or even to provide notification that its data has been discarded.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference and to de-allocate the returned data.

RELATED INFORMATION

Functions: **memory_object_data_supply**, **memory_object_server**.

memory_object_data_unlock

Server Interface — Requests access to a memory object

SYNOPSIS

kern_return_t **memory_object_data_unlock**

(mach_port_t	*abs_mem_port,*
mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
vm_size_t	*length,*
vm_prot_t	*desired_access*);

seqnos_memory_object_data_unlock

Sequence Number form

kern_return_t **seqnos_memory_object_data_unlock**

(mach_port_t	*abs_mem_port,*
mach_port_seqno_t	*seqno*,
mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
vm_size_t	*length,*
vm_prot_t	*desired_access*);

DESCRIPTION

A **memory_object_data_unlock** function is called as the result of a kernel message requesting the memory manager to permit at least the desired access to the specified data cached by the kernel. The memory manager is expected to use the **memory_object_lock_request** call in response.

PARAMETERS

abs_mem_port
: [in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

seqno
: [in scalar] The sequence number of this message relative to the abstract memory object port.

mem_cache_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. The offset must be on a page boundary.

length

[in scalar] The number of bytes to which the access applies, starting at *offset*. The number converts to an integral number of memory object pages.

desired_access

[in scalar] The memory access modes requested for the cached data. Possible values are obtained by or'ing together the following values:

VM_PROT_READ
Allows read access.

VM_PROT_WRITE
Allows write access.

VM_PROT_EXECUTE
Allows execute access.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

Functions: **memory_object_lock_completed**, **memory_object_lock_request**, **memory_object_server**.

memory_object_lock_completed

Server Interface — Indicates completion of a consistency control call

SYNOPSIS

```
kern_return_t memory_object_lock_completed
                (mach_port_t                    reply_port,
                mach_port_t                     mem_cache_control,
                vm_offset_t                     offset,
                vm_size_t                       length);
```

seqnos_memory_object_lock_completed

Sequence Number form

```
kern_return_t seqnos_memory_object_lock_completed
                (mach_port_t                    reply_port,
                mach_port_seqno_t               seqno,
                mach_port_t                     mem_cache_control,
                vm_offset_t                     offset,
                vm_size_t                       length);
```

DESCRIPTION

A **memory_object_lock_completed** function is called as the result of a kernel message confirming the kernel's action in response to a **memory_object_lock_request** call from the memory manager. The memory manager can use the **memory_object_lock_request** call to:

- Alter access restrictions specified in the **memory_object_data_supply** call or a previous **memory_object_lock_request** call.
- Write back modifications made in memory.
- Invalidate its cached data.

When the kernel completes the requested actions, it calls **memory_object_lock_completed** (asynchronously) using the port explicitly provided in the **memory_object_lock_request** call. Because the memory manager cannot know which pages have been modified, or even which pages remain in the cache, it cannot know how many pages will be written back in response to a **memory_object_lock_request** call. Receiving the **memory_object_lock_completed** call is the only sure means of detecting completion. The completion call includes the offset and length values from the consistency request to distinguish it from other consistency requests.

PARAMETERS

reply_port
: [in reply (receive) right] The port supplied in the corresponding **memory_object_lock_request** call.

seqno

[in scalar] The sequence number of this message relative to the port named in the **memory_object_lock_completed** message.

mem_cache_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. The offset must be on a page boundary.

length

[in scalar] The number of bytes to which the call refers, 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 will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

Functions: **memory_object_lock_request**, **memory_object_server**.

memory_object_notify

Server Interface — Announces an attempt by a kernel to establish a mapping via a memory object representative

SYNOPSIS

kern_return_t **memory_object_notify**

(mach_port_t	*mem_obj_rep,*
mach_port_t	*mem_cache_control,*
mach_port_t	*host_name,*
vm_size_t	*page_size*);

seqnos_memory_object_notify

Sequence Number form

kern_return_t **seqnos_memory_object_notify**

(mach_port_t	*mem_obj_rep,*
mach_port_seqno_t	*seqno*,
mach_port_t	*mem_cache_control,*
mach_port_t	*host_name,*
vm_size_t	*page_size*);

DESCRIPTION

A **memory_object_notify** function is called as the result of a kernel message notifying a memory manager that the kernel has been asked to map the specified memory object representative into a task's virtual address space.

When asked to map a memory object for the first time, the kernel responds by making a **memory_object_notify** call on the memory object representative. This call is the first message in a hand-shake dialog authenticating the memory manager (the receiver for the representative port) and the kernel to each other, thereby establishing the identity and properties of the underlying memory object.

The call supplies the memory manager with the kernel generated memory cache control port, which the memory manager must supply in its subsequent **memory_object_establish** call to authenticate itself to the kernel and which the memory manager will use to manipulate the memory cache object once initialization is complete. The supplied *host_name* port names the kernel mapping the object. The memory manager uses this port to verify the identity of the kernel sending the **memory_object_notify** message. This port is the destination of the subsequent **memory_object_establish** call.

The kernel holds send rights for the memory object representative port, and both send and receive rights for the memory cache control port.

The memory cache control port is initially created with default attributes. It (as well as the associated memory cache object) will be assigned its proper attributes as a result of the **memory_object_establish** call. No messages should be sent to the memory cache control port until memory object initialization is complete.

The call also supplies the virtual page size to be used for the memory mapping. The memory manager can use this size to detect mappings that use different data structures at initialization time, or to allocate buffers for use in reading data.

If a memory object is mapped into the address space of more than one task on different hosts (with independent kernels), the memory manager will receive a **memory_object_notify** call from each kernel, containing a unique set of host and object ports. Note that each kernel may also use a different page size.

PARAMETERS

mem_obj_rep
: [in memory-object-representative (receive) right] The memory object representative port that represents the memory object data, as supplied to the kernel in a **vm_map** call.

seqno
: [in scalar] The sequence number of this message relative to the memory object representative port.

mem_cache_control
: [in memory-cache-control send right] A send right to the kernel generated memory cache control port to be used in the **memory_object_establish** reply.

host_name
: [in host-name send right] The name port for the host requesting the memory object initialization. (The function **mach_host_self** returns send rights to the host's name port.)

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 and offsets that fall on page boundaries.

NOTES

When the memory manager is ready to manage this object, it verifies that the supplied host name port does name a host and then responds to this port with a **memory_object_establish** call.

By virtue of the fact that the kernel sent this message, it did not know the association of the memory object representative to the underlying abstract memory object. If the abstract object underlying the memory representative is currently initialized, the manager should respond with **memory_object_representative** instead, since the supplied memory cache control port does not name the memory cache object for the abstract memory object.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the cache control and host name port references.

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_representative**, **memory_object_terminate**, **memory_object_server**.

memory_object_rejected

Server Interface — Indicates that a **memory_object_establish** message was discarded

SYNOPSIS

kern_return_t **memory_object_rejected**
(mach_port_t *abs_mem_port,*
mach_port_t *rejected_cache_control*,
kern_return_t *reason*);

seqnos_memory_object_rejected

Sequence Number form

kern_return_t **seqnos_memory_object_rejected**
(mach_port_t *abs_mem_port,*
mach_port_seqno_t *seqno,*
mach_port_t *rejected_cache_control*,
kern_return_t *reason*);

DESCRIPTION

A **memory_object_rejected** function is called as the result of a kernel message notifying a memory manager that a **memory_object_establish** message from the manager was ignored. This occurs when the manager believes it has finished initializing a memory object that either was not being initialized (a client faked a **memory_object_notify** message) or the kernel no longer wishes to initialize the object. The kernel makes this call to allow the memory manager to clean up data structures associated with the invalid mappings. In such cases, the *memory_control* port sent isn't really a memory cache control port. The kernel returns send rights to this port as a means of identifying which **memory_object_establish** message was discarded.

PARAMETERS

abs_mem_port
: [in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

seqno
: [in scalar] The sequence number of this message relative to the abstract memory object port.

rejected_cache_control
: [in random send right] A port that the memory manager thought was a memory cache control port. If the memory object has been supplied to

more than one kernel, this parameter identifies the kernel that is making the call.

reason
[in scalar] The reason the memory object initialization failed.

KERN_INVALID_MEMORY_CONTROL
The memory cache control port supplied in the corresponding **memory_object_establish** message was not a memory cache control port.

KERN_INVALID_LEDGER
The ledger port supplied in the corresponding **memory_object_establish** message was not a ledger port.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the random port reference.

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_notify**, **memory_object_server**.

memory_object_supply_completed

Server Interface — Indicates completion of a data supply call

SYNOPSIS

kern_return_t **memory_object_supply_completed**

(mach_port_t	*reply_port,*
mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
vm_size_t	*length*,
kern_return_t	*result*,
vm_offset_t	*error_offset*);

seqnos_memory_object_supply_completed

Sequence Number form

kern_return_t **seqnos_memory_object_supply_completed**

(mach_port_t	*reply_port,*
mach_port_seqno_t	*seqno*,
mach_port_t	*mem_cache_control,*
vm_offset_t	*offset,*
vm_size_t	*length*,
kern_return_t	*result*,
vm_offset_t	*error_offset*);

DESCRIPTION

A **memory_object_supply_completed** function is called as the result of a kernel message confirming the kernel's action in response to a **memory_object_data_supply** call from the memory manager.

When the kernel accepts the pages, it calls **memory_object_supply_completed** (asynchronously) using the port explicitly provided in the **memory_object_data_supply** call. Because the data supply call can provide multiple pages, not all of which the kernel may necessarily accept and some of which the kernel may have to return to the manager (if precious), the kernel provides this response. If the kernel does not accept all of the pages in the data supply message, it will indicate so in the completion response. If the pages not accepted are precious, they will be returned (in **memory_object_data_return** messages) before it sends this completion message. The completion call includes the offset and length values from the supply request to distinguish it from other supply requests.

PARAMETERS

reply_port
: [in reply (receive) right] The port supplied in the corresponding **memory_object_data_supply** call.

seqno

[in scalar] The sequence number of this message relative to the port named in the **memory_object_data_supply** call.

mem_cache_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 from the corresponding data supply call. The offset must be on a page boundary.

length

[in scalar] The number of bytes accepted. The number converts to an integral number of memory object pages.

result

[in scalar] A kernel return code indicating the result of the supply operation, possibly KERN_SUCCESS. KERN_MEMORY_PRESENT is currently the only recoverable error returned; other errors (invalid arguments, for example) abort the data supply operation.

error_offset

[in scalar] The offset within the memory object where the first error occurred.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

Functions: **memory_object_data_supply**, **memory_object_server**.

memory_object_terminate

Server Interface — Relinquishes access to a memory object

SYNOPSIS

kern_return_t **memory_object_terminate**

(mach_port_t	*abs_mem_port,*
mach_port_t	*mem_cache_control*);

seqnos_memory_object_terminate

Sequence Number form

kern_return_t **seqnos_memory_object_terminate**

(mach_port_t	*abs_mem_port,*
mach_port_seqno_t	*seqno,*
mach_port_t	*mem_cache_control*);

DESCRIPTION

A **memory_object_terminate** function is called as the result of a kernel message notifying a memory manager that no mappings of the specified memory object remain. The kernel makes this call to allow the memory manager to clean up data structures associated with the de-allocated mappings. The kernel relinquishes its rights for all memory object ports.

The kernel terminates a memory object only after all address space mappings of the object have been de-allocated, or upon explicit request by the memory manager via **memory_object_destroy**.

PARAMETERS

abs_mem_port
: [in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

seqno
: [in scalar] The sequence number of this message relative to the abstract memory object port.

mem_cache_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.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

Functions: **memory_object_destroy**, **memory_object_notify**, **mach_port_deallocate**, **memory_object_server**.

CHAPTER 13 Privileged Interfaces

This chapter discusses “privileged” kernel calls — those that can affect the protection of the system.

add_partition

Function — Adds a partition to the default memory manager's backing store.

SYNOPSIS

```
kern_return_t add_partition
        (mach_port_t        service,
         device_t           device,
         recnum_t           recnum,
         vm_size_t          recsize,
         vm_size_t          length);
```

DESCRIPTION

The **add_partition** function provides the identity of backing space to the default memory manager.

PARAMETERS

service
: [in default-pager send right] The service port to the default memory manager, as returned by **bootstrap_ports**.

device
: [in device send right] The port naming the backing store device.

recnum
: [in scalar] The record number of the first record to use.

recsize
: [in scalar] The size, in bytes, of each record on the device.

length
: [in scalar] The total size of the partition, in bytes.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**, **bootstrap_ports**.

bootstrap_arguments

Function — Obtain arguments used by the bootstrap task to invoke servers

SYNOPSIS

kern_return_t **bootstrap_arguments**
(mach_port_t *bootstrap,*
boot_args_t *arguments,*
mach_msg_type_number_t* *argumentsCnt*);

DESCRIPTION

The **bootstrap_arguments** function obtains the arguments that were used by the bootstrap task to invoke servers.

PARAMETERS

bootstrap
[in bootstrap send right] A bootstrap port, the host's or that obtained by **bootstrap_ports**.

arguments
[out string] The argument string provided in the "call" of the user task at its invocation by the bootstrap task.

argumentsCnt
[out scalar] The number of space-separated items in *arguments*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **bootstrap_ports**

bootstrap_ports

Function — Obtains selected system privilege ports from the kernel or a bootstrap task

SYNOPSIS

kern_return_t **bootstrap_ports**
(mach_port_t *bootstrap,*
boot_port_mask_t* *port_mask,*
mach_port_array_t *boot_ports*);

DESCRIPTION

The **bootstrap_ports** function returns send rights to the requested special ports, as indicated by *port_mask*.

PARAMETERS

bootstrap
[in bootstrap send right] The bootstrap port obtained by **task_get_bootstrap_port**.

port_mask
[pointer to in/out scalar] A mask obtained by a bitwise OR of the following values:

GET_DEFAULT_PAGER_PORT
Return the default pager's service port (not applicable to the kernel; applicable to the system bootstrap task)

GET_DEVICE_MASTER_PORT
Return the device master port; used with **device_open**.

GET_HOST_CONTROL_PORT
Return the host control port; used to obtain other privileged ports and information.

GET_HOST_PAGING_PORT
Return the host paging port; used with **vm_set_default_memory_manager**.

GET_HOST_SECURITY_PORT
Return the host security port; used for **task_create_security_id** and **host_security_id_list**.

GET_LEDGER_CONTROL_PORT
Return the control port for the ledger containing the requesting task's resources (not applicable to the kernel).

boot_ports
[out array of send rights] The returned port right, indexed as follows (all unreturned rights are NULL):

DEFAULT_PAGER_PORT_INDEX

DEVICE_MASTER_PORT_INDEX

HOST_CONTROL_PORT_INDEX

HOST_PAGING_PORT_INDEX

HOST_SECURITY_PORT_INDEX

LEDGER_CONTROL_PORT_INDEX

RETURN VALUE

Only generic errors apply. The port mask is adjusted at output to indicate the ports actually delivered.

RELATED INFORMATION

Functions: **device_open**, **host_get_clock_control**, **host_processor_set_priv, task_create_security_id**, **host_security_id_list**, **task_get_special_port, vm_set_default_memory_manager**.

device_open

Function — Establish a connection to a device.

SYNOPSIS

```
kern_return_t device_open
                (mach_port_t                    master_port,
                 mach_port_t                    ledger,
                 dev_mode_t                     mode,
                 security_id_t                  security_id,
                 dev_name_t                     name,
                 mach_port_t*                   device_port);
```

device_open_request

Asynchronous Function form — Asynchronously request a connection to a device

```
kern_return_t device_open_request
                (mach_port_t                    master_port,
                 mach_port_t                    reply_port,
                 mach_port_t                    ledger,
                 dev_mode_t                     mode,
                 security_id_t                  security_id,
                 dev_name_t                     name);
```

ds_device_open_reply

Asynchronous Server Interface form — Receive the reply from an asynchronous open

```
kern_return_t ds_device_open_reply
                (mach_port_t                    reply_port,
                 kern_return_t                  device_code,
                 mach_port_t                    device_port,
                 mach_msg_type_number_t         device_portPoly);
```

DESCRIPTION

The **device_open** function opens a device object. The open operation of the device driver is invoked. 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 device master port. This port is provided to/by 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.

ledger
[in ledger-name send right] Resource ledger (on the destination host) from which the device will draw its resources.

mode
[in scalar] Opening mode. This is the bit-wise OR of the following values:

D_EXCLUSIVE
Exclusive use of the device

D_NODELAY
Do not delay on open

D_READ
Read access

D_VOLUME
Allow access to volume label and beyond disk partition limits

D_WRITE
Write access

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.

device_code
[in scalar] Status of the open.

device_port
[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 *device_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.

KERN_INVALID_LEDGER
ledger was not a kernel ledger.

RELATED INFORMATION

Functions: **device_close**, **device_reply_server**.

host_get_boot_info

Function — Return operator boot information

SYNOPSIS

kern_return_t **host_get_boot_info**
(mach_port_t *host_priv*,
kernel_boot_info_t *boot_info*);

DESCRIPTION

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

PARAMETERS

host_priv
[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 information.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**.

host_get_clock_control

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

SYNOPSIS

```
kern_return_t host_get_clock_control
        (mach_port_t        host_priv,
         clock_id_t         clock_id,
         mach_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.

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**.

host_processor_set_priv

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

SYNOPSIS

kern_return_t **host_processor_set_priv**
(mach_port_t *host_priv,*
mach_port_t *set_name,*
mach_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: **processor_set_default, processor_set_policy_control**.

host_reboot

Function — Reboot this host

SYNOPSIS

```
kern_return_t host_reboot
        (mach_port_t          host_priv,
         host_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.

host_security_id_list

Function — Returns a list of the active security IDs

SYNOPSIS

kern_return_t **host_security_id_list**

(mach_port_t	*host_security,*
security_id_array_t	*security_id_list,*
mach_msg_type_number_t*	*security_id_listCnt*);

DESCRIPTION

The **host_security_id_list** function returns a list of the active security IDs within *host_security*. A security ID is active if at least one existing task has that security ID.

PARAMETERS

host_security
: [in security send right] The security port for the host for which the security ID list is to be returned, obtained via **bootstrap_ports**.

security_id_list
: [out array of security IDs] The returned list of security IDs, in no particular order.

security_id_listCnt
: [in/out scalar] On input, the size of *security_id_list*; on output, the returned count of security IDs in *security_id_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create_security_id**, **mach_msg**, **bootstrap_ports**.

host_statistics

Function — Returns statistics for a host

SYNOPSIS

kern_return_t **host_statistics**

(mach_port_t	*host_priv,*
host_flavor_t	*flavor,*
host_info_t	*host_info_out*,
mach_msg_type_number_t*	*host_info_outCnt*);

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
: [out structure] Statistics about the specified host.

host_info_outCnt
: [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**.

ledger_create

Function — Create a new ledger

SYNOPSIS

kern_return_t **ledger_create**

(mach_port_t	*host_priv*,
resource_usage_t	*memory_entitlement,*
resource_ratio_t	*memory_ratio*,
resource_usage_t	*virtual_entitlement,*
resource_ratio_t	*virtual_ratio*,
mach_port_t*	*ledger_control*,
mach_port_t*	*ledger_name*);

DESCRIPTION

The **ledger_create** function creates a new ledger with specified resource limits.

PARAMETERS

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

memory_entitlement
: [in scalar] The amount of memory (in bytes) that can be used regardless of system load (applies to wired memory).

memory_ratio
: [in scalar] The ratio of system resources granted (applies to wired memory).

virtual_entitlement
: [in scalar] The amount of memory (in bytes) that can be used regardless of system load (applies to address space size).

virtual_ratio
: [in scalar] The ratio of system resources granted (applies to address space size).

ledger_control
: [out ledger-control send right] The control port for the new ledger.

ledger_name
: [out ledger-name send right] The name port for the new ledger.

NOTES

The kernel sporadically checks all of the system ledgers. If the resource consumption associated with a ledger exceeds the specified entitlement and the ratio of system resources consumed exceeds the specified ratio, then the ledger is considered for destruction, destroying all objects bound to it.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **ledger_terminate**, **task_create**, **memory_object_establish**, **device_open**, **clock_alarm**.

task_create_security_id

Function — Creates a new task with an explicit security ID

SYNOPSIS

kern_return_t **task_create_security_id**

(mach_port_t	*host_security,*
mach_port_t	*ledger_name*,
mach_port_t	*parent_task,*
security_id_t	*security_id*,
boolean_t	*inherit_memory,*
mach_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

host_security
: [in security send right] The security port for the host, obtained via **bootstrap_ports**.

ledger_name
: [in ledger-name send right] Resource ledger (on the destination host) from which the task will draw its resources.

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

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

inherit_memory
: [in boolean] 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_LEDGER
: *ledger* does not name a ledger on the host on which *task* lies.

KERN_INVALID_TASK
: *parent_task* does not name a task.

RELATED INFORMATION

Functions: **task_create**, **mach_msg**, **host_security_id_list**, **bootstrap_ports**.

thread_wire

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

SYNOPSIS

```
kern_return_t thread_wire
        (mach_port_t        host_priv,
         mach_port_t        target_thread,
         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

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

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

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

RETURN VALUE

KERN_INVALID_THREAD
: *target_thread* does not name a thread.

RELATED INFORMATION

Functions: **vm_wire**.

vm_set_default_memory_manager

Function — Sets the default memory manager.

SYNOPSIS

kern_return_t **vm_set_default_memory_manager**
(mach_port_t *virtual_paging_object,*
mach_port_t *backing_space*);

DESCRIPTION

The **vm_set_default_memory_manager** function establishes the abstract backing space object (the default memory manager) for a host.

PARAMETERS

virtual_paging_object
[in virtual-paging-object send right] The port naming the kernel's virtual paging object, obtained via **bootstrap_ports**.

backing_space
[in abstract-backing-space send right] A memory manager port naming the abstract backing space object to which anonymous pages will be sent (actually, the backing space object to which the kernel will send page-in and page-out requests).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **vm_allocate**, **vm_set_recycle_limit**, **add_partition**, **bootstrap_ports**.

vm_set_recycle_limit

Function — Adjust kernel's paging space recycle threshold

SYNOPSIS

kern_return_t **vm_set_recycle_limit**
(mach_port_t *virtual_paging_object,*
vm_size_t *recycle_limit*);

DESCRIPTION

The **vm_set_recycle_limit** function establishes the kernel's backing space recycle threshold.

The allocation of blocks in the virtual backing space is performed by the kernel. The kernel has a tendency to stream data out, allocating further and further into the backing space. Setting the recycle limit affects the threshold before the kernel starts reallocating space.

PARAMETERS

virtual_paging_object
[in virtual-paging-object send right] The port naming the kernel's virtual paging object, obtained via **bootstrap_ports**.

recycle_limit
[in scalar] The new recycle limit, in bytes.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**, **bootstrap_ports**.

CHAPTER 14 Protected Interfaces

This chapter discusses protected, or restricted, interfaces. Use of these interfaces does not affect the protection afforded by the system but do affect the operation of the system.

clock_set_attributes

Function — Sets attributes of a clock

SYNOPSIS

```
kern_return_t clock_set_attributes
        (mach_port_t                    clock_ctrl,
         clock_flavor_t                 flavor,
         clock_attr_t                   clock_attr,
         mach_msg_type_number_t         clock_attrCnt);
```

DESCRIPTION

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

PARAMETERS

clock_ctrl
: [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 (the smallest interval between clock ticks) and not greater than the maximum resolution of the clock.

clock_attr
: [pointer to in scalar] New attribute.

clock_attrCnt
: [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**.

clock_set_time

Function — Sets the current time

SYNOPSIS

kern_return_t **clock_set_time**
(mach_port_t *clock_ctrl,*
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 time-stamps set to the current clock time just prior to the new time being set with a return code of KERN_ABORTED.

PARAMETERS

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

new_time
[in structure] The 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**, **alarm_set**, **clock_sleep**.

Data Structures: **tvalspec**.

ledger_terminate

Function — Destroy a ledger

SYNOPSIS

kern_return_t **ledger_terminate**
(mach_port_t *target_ledger*);

DESCRIPTION

The **ledger_terminate** function destroys a ledger. All resources drawn from this ledger will also be destroyed.

PARAMETERS

target_ledger
[in ledger-control send right] The ledger to destroy.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **ledger_create**.

processor_set_policy_control

Function — Controls the scheduling attributes governing the processor set

SYNOPSIS

```
kern_return_t processor_set_policy_control
        (mach_port_t                  processor_set_control,
        processor_set_flavor_t        flavor,
        processor_set_info_t          policy_info,
        mach_msg_type_number_t        policy_infoCnt,
        boolean_t                     change_task_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_infoCnt
[in scalar] The size of the buffer (in natural-sized units).

change_task_threads
[in boolean] 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_default**, **processor_set_info**.

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

processor_set_statistics

Function — Returns scheduling statistics for a processor set.

SYNOPSIS

```
kern_return_t processor_set_statistics
        (mach_port_t                 processor_set_control,
         processor_set_flavor_t      flavor,
         processor_set_info_t        pset_info,
         mach_msg_type_number_t*     pset_infoCnt);
```

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**.

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

pset_infoCnt
: [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**.

processor_set_tasks

Function — Returns a list of the tasks bound to a processor set

SYNOPSIS

kern_return_t **processor_set_tasks**

(mach_port_t	*processor_set_control,*
mach_port_array_t	*tasks,*
mach_msg_type_number_t *	*tasksCnt*);

DESCRIPTION

The **processor_set_tasks** function returns a list of all tasks currently bound to the processor set.

PARAMETERS

processor_set_control
: [in processor-set-control send right] The control port for the processor set.

tasks
: [out array of task send rights] The returned list of tasks within *processor_set_control*, in no particular order.

tasksCnt
: [in/out scalar] On input, the size of *tasks*; on output, the returned count of tasks in *tasks*. The maximum size of *tasks* is PSET_TASKS_MAX.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_set_policy**.

task_set_policy

Function — Sets the scheduling policy to apply to a task

SYNOPSIS

```
kern_return_t task_set_policy
        (mach_port_t              processor_set_control,
         mach_port_t              task,
         policy_t                 policy,
         policy_base_t            base,
         mach_msg_type_number_t   baseCnt,
         policy_limit_t           limit,
         mach_msg_type_number_t   limitCnt,
         boolean_t                change);
```

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

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

task
: [in task send right] The task whose 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 policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

baseCnt
: [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**.

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

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

RETURN VALUE

KERN_INVALID_TASK
: *task* is not a task in processor set *processor_set_control*.

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**.

thread_set_policy

Function — Sets the scheduling policy to apply to a thread

SYNOPSIS

kern_return_t **thread_set_policy**

(mach_port_t	*processor_set_control,*
mach_port_t	*thread,*
policy_t	*policy,*
policy_base_t	*base*,
mach_msg_type_number_t	*baseCnt*,
policy_limit_t	*limit*,
mach_msg_type_number_t	*limitCnt*);

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

processor_set_control
[in processor-set-control send right] The control port for the processor set to which the thread is currently assigned.

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 policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

baseCnt
[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**.

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

RETURN VALUE

KERN_INVALID_THREAD
thread is not a thread in processor set *processor_set_control*.

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**.

APPENDIX A

Intel 386/486 Platform Device Interface Specifics

This appendix supplements the Kernel Device Interface chapter by providing specific argument values and constraints associated with the device drivers available for the Intel 386/486 reference platform.

Device Names

The general form for MK++ device names is **aabc**, where **aa** represents the device class, **b** is the unit number within that class, and **c** is an optional subunit or partition identifier. The unit number is numeric, starting at zero; the subunit/partition is alphabetic, starting with "`a`" and limited by the respective device driver. Most physical devices carry at least two such names, the first of which is computed from the device driver (e.g. "`sd`" for SCSI Disk) and unit number. An alias is also assigned by generic class (e.g. "`disk`") with a system-wide sequential unit number.

For the purposes of the **device_open** call, **b** defaults to zero and **c** defaults to null (implying the first subunit). As a result, the first partition of the first configured disk drive may be referred to as "`disk`", "`disk0`", or "`disk0a`" interchangeably.

Device Concurrency and Covert Storage Channels

While the MK++ kernel is explicitly *not* the reference monitor for device access, it is called upon to enforce some per-request constraints upon shared devices. The general requirement is that an untrusted reader of a shared device shall not be capable of modifying device state in any way that is detectable by another reader of the same device. For this reason, certain operations that might otherwise be permissible to a reader now require one or both of the new `D_EXCLUSIVE` or `D_VOLUME` flags to have been specified to **device_open**. Some devices are necessarily unshareable among concurrent users, which simplifies such checks in those cases.

Console, Keyboard, and Mouse Drivers

The i386/486 physical console is named "`kd0`" and is configured as the first device on the system. It acquires "`tty0`" as its alias by virtue of its character-mode operation. When the `cons_is_com1` boot flag is *not* set, the special "`console`" alias is also applied. The (raw) keyboard, PS2 mouse, and PORT mouse devices are assigned "`kd0b`", "`kd0c`", and "`kd0d`", respectively.

In addition to being identified as console subaddresses, the keyboard and various mouse devices have been given the following aliases for use by the X server:

- "`kbd0`" is the console keyboard in "raw" mode.
- "`kbd2`" is a PS2 mouse, if configured.
- "`kbd3`" is a port mouse, if configured.
- "`kbd4`" is a (two-button) Microsoft® mouse on COM1.
- "`kbd5`" is a (three-button) Mouse Systems® mouse on COM1.
- "`kbd6`" is a (two-button) Microsoft® mouse on COM2.
- "`kbd7`" is a (three-button) Mouse Systems® mouse on COM2.

The console driver is configured at physical and port address 0x60 and is assigned Processor Interrupt Channel 1. Console interrupts are handled at the `SPL_TTY` level.

Open Modes and Concurrency

The "raw" keyboard (`kd0b` or `kb0d`) is opened exclusively, as is whichever mouse is configured. The console display is sharable among multiple writers.

Opening modes `D_WRITE` and `D_READ` may be specified with all subdevices; no checks are made during the processing of **device_open**. Opening mode `D_EXCLUSIVE` should *not* be used with the physical console or its subdevices. Mode `D_VOLUME` is not meaningful, but will not be rejected.

Get Status Request Flavors

The following `device_flavor_t` values apply to the physical console/keyboard:

- `KDGKBDTYPE` returns the constant `KB_VANILLAKB`. It may only be specified on calls to the "raw" console keyboard device.
- `KDGKBENT` gets an entry from the current keyboard mapping table.
- `KDGSTATE` returns the current state of the keyboard as an integer.

See the section following for additional flavors that may be passed through to mouse devices connected to serial ports.

Set Status Request Flavors

The following `device_flavor_t` values apply to the physical console/keyboard:

- `KDSETBELL` enables/disables the speaker port, which produces the audible "beep" in response to `BEL` characters sent to the display.
- `KDSKBDMODE` sets the mode of the console keyboard to either `KB_EVENT` or `KB_ASCII`. It may only be specified on calls to the "raw" console keyboard device.
- `KDSKBENT` sets an entry in the current keyboard mapping table. `D_WRITE` mode is required.

See the section following for additional flavors that may be passed through to mouse devices connected to serial ports.

Unsupported Calls

Neither the **device_map** nor the **device_set_filter** calls are supported by the console driver.

Serial Communications (TTY) Drivers

Each serial communications port is named sequentially beginning with "`com0`" and has an alias of "`tty#`", where `#` is assigned sequentially to each configured tty device.

The serial communications ports are configured at port addresses 0x3f8, 0x2f8 0x3e8, and 0x2e8. They are assigned Processor Interrupt Channels 4, 3, 5, and 12, respectively. Note that the PICs for `com2` and `com3` can conflict with those of the configured Ethernet controller, below. Interrupts are handled at the `SPL_TTY` level.

Get Status Request Flavors

The following `device_flavor_t` values apply to character devices, including serial mouse devices associated with the system console.

- `TTY_MODEM` obtains modem control information for the serial port.
- `TTY_OUT_OF_BAND` reads out-of-band events from the serial device.
- `TTY_STATUS_COMPAT` obtains information about the serial device.
- `TTY_STATUS_NEW` obtains information about the serial device.

Set Status Request Flavors

The following `device_flavor_t` values apply to the character devices, including serial mouse devices associated with the system console.

- `TTY_DRAIN` discards queued output.
- `TTY_FLUSH` clears both input and output queues.
- `TTY_START` starts queued output.
- `TTY_STOP` stops output.
- `TTY_SET_BREAK` sets the break condition. `D_WRITE` mode is required.

- TTY_CLEAR_BREAK clears the break condition. D_WRITE mode is required.
- TTY_MODEM sets modem control information for the serial port. D_WRITE mode is required.
- TTY_NMODEM sets the soft carrier for the serial port. D_WRITE mode is required.
- KDSKBENT sets an entry in the current keyboard mapping table. D_WRITE mode is required. Applies only to the the console device when remapped onto COM1.
- TTY_STATUS_COMPAT sets special characters and speed for the serial device. D_WRITE mode is required.
- TTY_STATUS_NEW sets special characters and speed for the serial device. D_WRITE mode is required.

Unsupported Calls

Neither the **device_map** nor the **device_set_filter** calls are supported by the serial communications driver.

VGA Drivers

When configured, the single VGA monitor is known as “vga0”. For compatibility with the X server, the “iopl” alias is assigned to the graphics card to enable device mapping.

Four VGA drivers are provided, which are probed at kernel initialization in the order listed below. If any of the first three drivers recognizes its supported graphics card, that driver is configured. The default is the generic driver for any other VGA card.

- The PVGA driver supports various graphics cards based upon the Western Digital 90C00 and 90C10 chips.
- The ET4000 driver supports the Sigma Legend and similar graphics cards.
- The S3 driver supports graphics cards from S3, Number 9 and Actix that use the S3 chip set.
- The generic VGA driver is used with all other graphics cards meeting basic VGA specifications.

The VGA controller is configured at port address 0x3b0 and physical address 0xa0000. It does not accept interrupts, so no Processor Interrupt Channel applies.

Device Map Specifics

Each VGA driver maps a 64KB window into the graphic card’s video RAM under the control of the VGA I/O registers. Access to the device memory is constrained to not exceed the mode specified to **device_open** — write access to the mapped memory requires write access to the device.

For the i386/486 platform, **device_map** of a device open in D_EXCLUSIVE or D_WRITE mode also sets up access to the corresponding i386/486 I/O registers. These registers are only usable while the memory object representative is mapped into the task’s address space; they are activated by **vm_map** and deactivated by **vm_deallocate** of the region or **device_close** of the mapped device.

The following I/O registers apply to all VGA devices:

0x200	0x3b0 - 0x3df

The S3 driver defines the following additional I/O registers:

0x42e8 - 9	0x4ae8 - 9	0x82e8 - 9	0x86e8 - 9
0x8ae8 - 9	0x8ee8 - 9	0x92e8 - 9	0x96e8 - 9
0x9ae8 - 9	0x9ee8 - 9	0xa2e8 - b	0xa6e8 - b
0xaae8 - b	0xaee8 - b	0xb2e8 - b	0xb6e8 - 9
0xbae8 - 9	0xbee8 - 9	0xe2e8 - b	

Unsupported Calls

The VGA drivers do not support either the inband or out-of-line forms of **device_read** or **device_write**, nor is **device_set_filter** supported. Calls to **device_get_status** and **device_set_status** are ignored and return success status without performing any action.

Network Drivers

Two network drivers are provided, which support multiple controllers of each type, subject to the requirement that each Processor Interrupt Channel must be uniquely assigned to one controller (and not conflict with a configured serial communications line, above). All interrupts are processed at the `SPL_BIO` interrupt level.

Up to three controllers based upon the PC586 chipset can be configured using physical addresses 0xd0000, 0xc0000 and 0xf0000. When present, these controllers are assigned Processor Interrupt Channels 9, 5, and 12, respectively. The assigned device names for Olivetti PC586 controllers are “`pc0`” through “`pc2`”.

Up to two controllers based upon the NS8390 chipset can be configured at port addresses 0x240, 0x280, 0x2a0, 0x2e0, or 0x300. When present, these controllers are assigned Processor Interrupt Channels 5 or 9 at start-up. The assigned device names for Western Digital 8003E EtherNet™ controllers are “`wd0`” and “`wd1`”, while the assigned device names for 3COM Etherlink 3c503 controllers are “`el0`” and “`el1`”.

Get Status Flavors

The following `device_flavor_t` values apply to the network devices.

- `NET_ADDRESS` returns an array of long integers that represent the network address of the device.
- `NET_STATUS` returns information on the network packet format used by this device.

Set Status Flavors

The following `device_flavor_t` value applies to the network devices.

- `NET_STATUS` sets selected flags to control the network device. The caller must have opened the device with `D_WRITE` permission.

Unsupported Calls

The network drivers do not support either the inband or out-of-line forms of **device_read**, since that functionality is performed through the use of device filters. The **device_map** service is also not supported.

Floppy Disk Driver

Floppy disk devices are named "`fd0`" and "`fd1`" when configured. Each drive is also assigned a "`disk#`" alias in the order of its recognition at system start-up. A subdevice suffix letter is used to indicate the media type, which defaults to "`a`".

- "`fd0a`" specifies a 3.5" DSDD (720KB) diskette.
- "`fd0b`" specifies a 3.5" DSHD (1.4MB) diskette.
- "`fd0c`" specifies a 5.25" DSSD (360KB) diskette.
- "`fd0d`" specifies a 5.25" DSHD (1.2MB) diskette.

The floppy controller is configured at port address 0x3f2 and assigned Processor Interrupt Channel 6. Floppy disk interrupts are handled at the `SPL_BIO` level.

Get Status Request Flavors

The following `device_flavor_t` values apply to floppy disk devices.

- `DEV_GET_SIZE` obtains the size of device blocks and the total size of the device.
- `V_GETPARMS` returns the media geometry information for the device.

Set Status Request Flavors

The following `device_flavor_t` values apply to floppy disk devices.

- `V_FORMAT` formats the volume. Caller must have `D_WRITE`, `D_EXCLUSIVE`, and `D_VOLUME` access to the device.
- `V_SETPARMS` causes the drive parameters to be rechecked. Caller must have `D_WRITE` access to the volume.
- `V_VERIFY` is provided for compatibility with other disk drivers but has no effect. Caller must have `D_WRITE` access to the volume.

Unsupported Calls

Neither the **device_map** nor the **device_set_filter** calls are supported by the floppy disk driver.

SCSI Subsystem

Both disk and tape device types are supported via Small Computer Systems Interface (SCSI) controllers. Two drivers have been provided for Adaptek AHA174x controllers in "enhanced" (EISA) mode and Adaptek AHA15xx (and compatible) controllers in "standard" (ISA) mode.

AHA15xx Controllers

Up to four 15xx-compatible ISA controllers can be configured, using port addresses 0x330, 0x334, 0x234, and 0x230. All share Processor Interrupt Channel 11 and process interrupts at the `SPL_BIO` interrupt level.

AHA174x Controllers

Up to sixteen 174x-compatible EISA controllers can be configured, based at physical addresses 0x0000 through 0xf000, incremented by 0x1000. All share Processor Interrupt Channel 12 and process interrupts at the `SPL_BIO` interrupt level.

SCSI Disk Devices

For each configured controller, eight disk devices are configured at SCSI addresses 0 through 7. Disk units are named "`sd#`", where `#` is computed as 16 * the controller number (starting at zero) + the SCSI address of the unit. Disk units may have up to 16 partitions, identified sequentially from "`a`" to "`p`". For compatibility with non-SCSI systems, each unit is given an alias of "`hd#`" to match "`sd#`". Each disk device (of any type) also receives a "`disk#`" alias, where `#` is assigned sequentially across the entire system. The hard disk boot partition also receives the "`boot_device`" alias.

Get Status Request Flavors

The following `device_flavor_t` values apply to SCSI disk devices.

- `DEV_GET_SIZE` obtains the size of device blocks and the total size of the device.
- `DIOCGDINFO` reads the disk label.
- `V_GETPARMS` returns the media geometry information for the device.
- `V_RDABS` reads data using the absolute block number set with `V_ABS`. Caller must have opened the device with `D_VOLUME` access.
- `V_VERIFY` read/verifies data using the absolute block number set with `V_ABS`. Caller must have opened the device with `D_VOLUME` access.

Set Status Request Flavors

The following `device_flavor_t` values apply to SCSI disk devices.

- `DIOCSDINFO` sets the in-memory image of the disk label. Caller must have opened the device with `D_VOLUME` and `D_WRITE` access.
- `DIOCSRETRIES` updates the retry threshold for bad block replacement.
- `DIOCWDINFO` writes the disk label. Caller must have opened the device with `D_VOLUME` and `D_WRITE` access.
- `DIOCWLABEL` write-enables/disables the label. Caller must have opened the device with `D_VOLUME` and `D_WRITE` access.
- `V_ABS` sets the absolute block number for the next `V_RDABS`, `V_VERIFY`, or `V_WRABS` request. Caller must have opened the device with `D_VOLUME` access.
- `V_REMOUNT` remounts the drive. Caller must have opened the device with `D_VOLUME` access.
- `V_WRABS` writes data using the absolute block number set with `V_ABS`. Caller must have opened the device with `D_VOLUME` and `D_WRITE` access.

SCSI Tape Devices

For each configured controller, eight tape devices are configured at SCSI addresses 0 through 7. Tape units are named "`st#`", where `#` is computed as 16 * the controller number (starting at zero) + the SCSI address of the unit. Each tape device (of any type) also receives a "`tape#`" alias, where `#` is assigned sequentially across the entire system.

Get Status Request Flavors

The following `device_flavor_t` values apply to SCSI tape devices.

- `MTIOCGET` gets extended status from the drive.
- `TAPE_STATUS` gets the current recording mode and mount information.

Set Status Request Flavors

The following `device_flavor_t` values apply to SCSI tape devices.

- `MTIOCTOP` performs a tape-specific operation.
- `TAPE_STATUS` sets the tape recording mode and mount control flags.

Unsupported Calls

Neither the **device_map** nor the **device_set_filter** calls are supported by the SCSI driver.

APPENDIX B

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 fields generated and interpreted by MIG stubs are 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 is also dictated by the MIG stub. For scalar rights, no additional information appears in the data stream. For arrays of port rights, 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 C Data Structures

This appendix discusses the specifics of the various structures used as a part of the kernel's various interfaces. This appendix does not discuss all of the various data types used by the kernel's interfaces, only the fields of the various structures used.

host_basic_info

Structure — Defines basic information about a host

SYNOPSIS

```
struct host_basic_info
{
        vm_size_t              max_cpus;
        vm_size_t              avail_cpus;
        vm_size_t              memory_size;
        cpu_type_t             cpu_type;
        cpu_subtype_t          cpu_subtype;
        vm_address_t           min_address;
        vm_address_t           max_address;
};
typedef struct host_basic_info*    host_basic_info_t;
```

DESCRIPTION

The **host_basic_info** structure defines the basic information available about a host.

FIELDS

max_cpus
: Maximum possible CPUs for which kernel is configured

avail_cpus
: Number of CPUs now available

memory_size
: Size of memory, in bytes

cpu_type
: CPU type

cpu_subtype
: CPU sub-type

min_address
: Minimum address in user space

max_address
: Maximum address in user space

NOTES

This structure is machine word length specific because of the memory size returned.

RELATED INFORMATION

Functions: **host_info.**

Data structures: **host_load_info**, **host_sched_info.**

host_load_info

Structure — Defines load information about a host

SYNOPSIS

```
#define CPU_STATE_USER          0
#define CPU_STATE_SYSTEM        1
#define CPU_STATE_IDLE          2
struct host_load_info
{
        load_t                  avenrun[3];
        load_t                  load_factor[3];
};
typedef struct host_load_info*  host_load_info_t;
```

DESCRIPTION

The **host_load_info** structure defines the loading information available about a host. The information returned is exponential averages over three periods of time: 5, 30 and 60 seconds.

FIELDS

avenrun
: Load average—average number of runnable processes divided by number of CPUs

load_factor
: The processing resources available to a new thread—the number of CPUs divided by (1 + the number of threads)

RELATED INFORMATION

Functions: **host_statistics.**

Data structures: **host_basic_info**, **host_sched_info.**

host_sched_info

Structure — Defines scheduling information about a host

SYNOPSIS

[1]	struct **host_sched_info**	
[2]	{	
[3]	mach_msg_timeout_t	*min_timeout*;
[4]	mach_msg_timeout_t	*min_quantum*;
[5]	};	
[6]	typedef struct host_sched_info*	**host_sched_info_t**;

DESCRIPTION

The **host_sched_info** structure defines the limiting scheduling information available about a host.

FIELDS

min_timeout
: Minimum time-out, in milliseconds

min_quantum
: Minimum quantum (period for which a thread can be scheduled if uninterrupted), in milliseconds

RELATED INFORMATION

Functions: **host_info.**

Data structures: **host_basic_info**, **host_load_info.**

mach_msg_descriptor

Structure — Defines the disposition of kernel processed message data

SYNOPSIS

```
typedef struct
{
        vm_address_t                 pad1;
        mach_msg_size_t              pad2;
        unsigned int                 pad3: 24;
        mach_msg_descriptor_type_t   type: 8;
} mach_msg_type_descriptor_t;
```

```
typedef struct
{
        mach_port_t                  name;
        mach_msg_size_t              pad1;
        unsigned int                 pad2: 16;
        mach_msg_type_name_t         disposition: 8;
        mach_msg_descriptor_type_t   type: 8;
} mach_msg_port_descriptor_t;
```

```
typedef struct
{
        vm_address_t                 address;
        mach_msg_size_t              size;
        boolean_t                    deallocate: 8;
        unsigned int                 pad1: 8;
        unsigned int                 pad2: 8;
        mach_msg_descriptor_type_t   type: 8;
} mach_msg_ool_descriptor_t;
```

```
typedef union
{
        mach_msg_type_descriptor_t   type;
        mach_msg_port_descriptor_t   port;
        mach_msg_ool_descriptor_t    out_of_line;
} mach_msg_descriptor_t;
```

DESCRIPTION

A **mach_msg_descriptor** structure describes the processing to be performed for an element of kernel processed data in a Mach message.

FIELDS

name
The name of the port whose right is being sent.

disposition
For port descriptors, the IPC processing to be done for the rights for the named port.

address
For out-of-line data descriptors, the address of the out-of-line data.

size
For out-of-line data descriptors, the size of the out-of-line region, in bytes.

deallocate
For out-of-line data descriptors, true if the set of pages containing the array should be de-allocated when the message is sent.

type
For any type of descriptor, the type of descriptor.

port
A descriptor that describes a single port right.

out_of_line
A descriptor that describes an out-of-line data array.

RELATED INFORMATION

Functions: **mach_msg**.

Data structures: **mach_msg_header**.

mach_msg_header

Structure — Defines the header portion for messages

SYNOPSIS

```
typedef struct
{
        mach_msg_descriptor_t       msgh_remote_port;
        mach_msg_descriptor_t       msgh_local_port;
        security_id_t               msgh_sender;
        mach_port_seqno_t           msgh_seqno;
        mach_msg_id_t               msgh_id;
        mach_msg_size_t             msgh_descriptor_count;
        mach_msg_size_t             msgh_size;
} mach_msg_header_t;
```

DESCRIPTION

The **mach_msg_header** structure defines the fixed size header of a Mach message. The header is followed by a message body containing data and port descriptors and zero or more data bytes.

Following the header (and any kernel processed descriptors), at natural alignment can be additional (un-typed) data, up to the size of the message (*msgh_size*). This extra data typically carries information used to decode the data stream and out-of-line data.

FIELDS

msgh_remote_port
: When sending, specifies the destination port of the message. The field must carry a legitimate send or send-once right for a port. When received, this field is swapped with *msgh_local_port*.

msgh_local_port
: When sending, specifies an auxiliary port right, which is conventionally used as a reply port by the recipient of the message. The field must carry a send right, a send-once right, MACH_PORT_NULL, or MACH_PORT_DEAD. When received, this field is swapped with *msgh_remote_port*.

msgh_sender
: The security ID of the sender of the message; the ID is set by the kernel).

msgh_seqno
: The sequence number of this message relative to the port from which it is received.

msgh_id

Not set or read by the **mach_msg** call. The conventional meaning is to convey an operation or function ID.

msgh_descriptor_count

The number of descriptors of kernel processed data (port rights and out-of-line data).

msgh_size

This field is ignored on send (the size to send is specified by the *send_size* parameter to **mach_msg**); the field is set on receive to the sum of the message header and body sizes (in bytes).

RELATED INFORMATION

Functions: **mach_msg**.

Data structures: **mach_msg_descriptor**.

mach_port_limits

**Structure — ** Defines resource limits for a port

SYNOPSIS

[1] struct **mach_port_limits**
[2] {
[3] mach_msg_type_number_t *memory_limit*;
[4] mach_msg_type_number_t *right_limit*;
[5] };
[6] typedef struct mach_port_limits* **mach_port_limits_t**;

DESCRIPTION

The **mach_port_limits** structure defines various limits governing the messages that can be sent through the port.

FIELDS

memory_limit
Limit on the number of bytes of out-of-line data that may be received in a message.

right_limit
Limit on the number of port rights that may be received in a message.

NOTES

There is no explicit limit on the total number of bytes for the message, but ledger resource limits apply.

RELATED INFORMATION

Functions: **mach_port_get_attributes**, **mach_port_set_attributes**.

mach_port_status

Structure — Defines information for a port

SYNOPSIS

[1] struct **mach_port_status**
[2] {
[3] mach_port_name_t *mps_pset*;
[4] };
[5] typedef struct mach_port_status **mach_port_status_t**;

DESCRIPTION

The **mach_port_status** structure defines information about a port.

FIELDS

mps_pset
Containing port set

NOTES

This structure is machine word length sensitive due to the presence of the port set name.

RELATED INFORMATION

Functions: **mach_port_get_attributes.**

mapped_tvalspec

Structure — Defines format of kernel maintained time in a mapped clock

SYNOPSIS

```
struct mapped_tvalspec
{
        tvalspec_t              mtv_time;
        unsigned int            mtv_csec;
};
typedef struct mapped_tvalspec  mapped_tvalspec_t;
```

DESCRIPTION

The **mapped_tvalspec** structure defines the format of the current-time structure maintained by the kernel and visible through a mapped clock (**clock_map_time**). The data in this structure is updated at the clock's current resolution and contains the same tvalspec value that would be returned by **clock_get_time**.

FIELDS

mtv_time
: Clock time.

mtv_csec
: A field used to synchronize with the kernel's setting of the time.

NOTES

Because of the race between the referencing of the multiple fields in the clock value and the kernel's setting them, they should be referenced as follows:

```
tvalspec_t*             ts;
mapped_tvalspec_t*      mtime;
do
{
        ts → tv_sec = mtime → mtv_time.tv_sec;
        ts → tv_nsec = mtime → mtv_time.tv_nsec;
} while (ts → tv_sec!= mtime → mtv_csec);
```

RELATED INFORMATION

Functions: **clock_map_time**, **clock_get_time.**

Data Structures: **tvalspec**.

memory_object_behave_info

Structure — Defines the behavioral attributes for a memory object

SYNOPSIS

```
[1] struct memory_object_behave_info
[2] {
[3]         memory_object_copy_strategy_t    copy_strategy;
[4]         boolean_t                        temporary;
[5] };
[6] typedef struct memory_object_behave_info* memory_object_behave_info_t;
```

DESCRIPTION

The **memory_object_behave_info** structure defines behavior relevant memory object attributes.

FIELDS

copy_strategy
: How the kernel should handle copying of regions associated with the memory object. The copy strategy cannot be changed once an object is initialized. Valid values are:

MEMORY_OBJECT_COPY_NONE
: Use normal procedure when copying the memory object's data. Normally, the kernel requests each page with read access, copies the data, and then (optionally) flushes the data.

MEMORY_OBJECT_COPY_DELAY
: Use copy-on-write technique. This strategy allows the kernel to efficiently copy large amounts of data and guarantees that the memory manager will not externally modify the data. It is the most commonly used copy strategy.

temporary
: When the last mapping of the memory object is removed, the object is destroyed without returning any in-memory pages.

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_get_attributes**, **memory_object_change_attributes**, **vm_region**, **vm_set_default_memory_manager**.

Data Structures: **memory_object_perf_info**.

memory_object_perf_info

Structure — Defines the performance attributes for a memory object

SYNOPSIS

[1] struct **memory_object_perf_info**
[2] {
[3] vm_size_t *cluster_size*;
[4] boolean_t *may_cache*;
[5] };
[6] typedef struct memory_object_perf_info* **memory_object_perf_info_t**;

DESCRIPTION

The **memory_object_perf_info** structure defines performance relevant memory object attributes.

FIELDS

cluster_size
Preferred cluster size (in bytes) for the memory object. This helps to determine how many pages are transferred in individual data request and return messages.

may_cache
Cache indicator. If true, the kernel can cache data associated with the memory object (keep the memory object active) even if no virtual memory references to it remain.

NOTES

Sharing cached data among all the clients of a memory object can have a major impact on performance, especially if it can be extended across successive, as well as concurrent, uses. For example, the memory objects that represent program images can be used regularly by different programs. By retaining the data for these memory objects in cache, the number of secondary storage accesses can be reduced significantly.

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_get_attributes**, **memory_object_change_attributes**, **vm_region**, **vm_set_default_memory_manager**.

Data Structures: **memory_object_behave_info**.

policy_fifo_info

Structure — Defines first-in-first-out scheduling information

SYNOPSIS

```
struct policy_fifo_base
{
        priority_t                    base_priority;
};

struct policy_fifo_limit
{
        priority_t                    max_priority;
};

struct policy_fifo_info
{
        priority_t                    max_priority;
        priority_t                    base_priority;
        boolean_t                     depressed;
};
```

DESCRIPTION

The **policy_fifo_info** structure defines the first-in-first-out scheduling policy information.

FIFO threads have two priorities associated with them by the system:

- A maximum priority value that can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set.
- A priority value that can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task.

FIELDS

base_priority
Scheduling priority

max_priority
Maximum scheduling priority

depressed
True if scheduling priority is depressed

RELATED INFORMATION

Functions: **thread_info**, **task_info**, **processor_set_info**, **processor_set_policy_control**, **task_policy**, **thread_policy**.

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

policy_rr_info

Structure — Defines round-robin scheduling information

SYNOPSIS

```
[1] struct policy_rr_base
[2] {
[3]         priority_t              base_priority;
[4]         mach_msg_timeout_t      quantum;
[5] };

[1] struct policy_rr_limit
[2] {
[3]         priority_t              max_priority;
[4] };

[1] struct policy_rr_info
[2] {
[3]         priority_t              max_priority;
[4]         priority_t              base_priority;
[5]         mach_msg_timeout_t      quantum;
[6]         boolean_t               depressed;
[7] };
```

DESCRIPTION

The **policy_rr_info** structure defines the round-robin scheduling policy information.

Round-robin threads have two priorities associated with them by the system:

- A maximum priority value that can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set.
- A priority value that can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task.

FIELDS

base_priority
Scheduling priority

quantum
Scheduling quantum (in milliseconds)

max_priority
Maximum scheduling priority

depressed
True if scheduling priority is depressed

RELATED INFORMATION

Functions: **thread_info**, **task_info**, **processor_set_info**, **processor_set_policy_control**, **task_policy**, **thread_policy**.

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

policy_timeshare_info

Structure — Defines timeshare scheduling information

SYNOPSIS

[1] struct **policy_timeshare_base**
[2] {
[3] priority_t *base_priority*;
[4] };

[1] struct **policy_timeshare_limit**
[2] {
[3] priority_t *max_priority*;
[4] };

[1] struct **policy_timeshare_info**
[2] {
[3] priority_t *max_priority*;
[4] priority_t *base_priority*;
[5] priority_t *cur_priority*;
[6] boolean_t *depressed*;
[7] };

DESCRIPTION

The **policy_timeshare_info** structure defines the timeshare scheduling policy information.

Timeshare threads have three priorities associated with them by the system:

- A priority value that can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task.
- A maximum priority value that can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set.
- A scheduled priority value that is used to make scheduling decisions for the thread. This value is determined on the basis of the user priority value by the scheduling policy (for timesharing, this means adding an increment derived from CPU usage).

FIELDS

max_priority
Maximum scheduling priority

base_priority
Base scheduling priority

cur_priority
Current scheduling priority

depressed
True if scheduling priority is depressed

RELATED INFORMATION

Functions: **thread_info**, **task_info**, **processor_set_info**, **processor_set_policy_control**, **task_policy**, **thread_policy**.

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

processor_set_basic_info

Structure — Defines the basic information about a processor set.

SYNOPSIS

```
[1] struct processor_set_basic_info
[2] {
[3]         cpu_number_t                    processor_count;
[4]         policy_t                        default_policy;
[5] };
[6] typedef struct processor_set_basic_info*    processor_set_basic_info_t;
```

DESCRIPTION

The **processor_set_basic_info** structure defines the basic information available about a processor set.

FIELDS

processor_count
Number of processors in this set

default_policy
Default policy to assign to threads whose otherwise assigned policy is not enabled

RELATED INFORMATION

Functions: **processor_set_info**.

Data Structures: **processor_set_load_info**.

processor_set_load_info

Structure — Defines the scheduling statistics for a processor set.

SYNOPSIS

```
struct processor_set_load_info
{
        mach_msg_type_number_t          task_count;
        mach_msg_type_number_t          thread_count;
        load_t                          load_average;
        load_t                          load_factor;
};
typedef struct processor_set_load_info*  processor_set_load_info_t;
```

DESCRIPTION

The **processor_set_load_info** structure defines the scheduling statistics maintained for a processor set.

FIELDS

task_count
: Number of tasks currently assigned to this processor set

thread_count
: Number of threads currently assigned to this processor set

load_average
: Average number of runnable processes divided by number of CPUs

load_factor
: The processing resources available to a new thread—the number of CPUs divided by (1 + the number of threads)

RELATED INFORMATION

Functions: **processor_set_statistics**.

Data Structures: **processor_set_basic_info**.

task_basic_info

Structure — Defines basic information for tasks

SYNOPSIS

```
[1]  struct task_basic_info
[2]  {
[3]          suspend_t                    suspend_count;
[4]          vm_size_t                    virtual_size;
[5]          vm_size_t                    resident_size;
[6]          tvalspec_t                   user_time;
[7]          tvalspec_t                   user_idle_time;
[8]          tvalspec_t                   system_time;
[9]          policy_t                     policy;
[10] };
[11] typedef struct task_basic_info*      task_basic_info_t;
```

DESCRIPTION

The **task_basic_info** structure defines the basic information array for tasks. The **task_info** function returns this array for a specified task.

FIELDS

suspend_count
: The current suspend count for the task.

virtual_size
: The number of virtual bytes for the task.

resident_size
: The number of resident bytes for the task

user_time
: The total user run time for terminated threads within the task.

user_idle_time
: The total user run time with scheduling depressed for terminated threads within the task.

system_time
: The total system run time for terminated threads within the task.

policy
: Default scheduling policy to apply to new threads.

NOTES

This structure is machine word length sensitive due to the presence of the virtual address sizes.

RELATED INFORMATION

Functions: **task_info**.

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

task_thread_times_info

Structure — Defines thread execution times information for tasks

SYNOPSIS

```
[1] struct task_thread_times_info
[2] {
[3]         tvalspec_t                    user_time;
[4]         tvalspec_t                    user_idle_time;
[5]         tvalspec_t                    system_time;
[6] };
[7] typedef struct task_thread_times_info*   task_thread_times_info_t;
```

DESCRIPTION

The **task_thread_times_info** structure defines thread execution time statistics for tasks. The **task_info** function returns these times for a specified task. The **thread_info** function returns this information for a specific thread.

FIELDS

user_time
: Total user run time for live threads.

user_idle_time
: The total user run time with scheduling depressed for terminated threads within the task.

system_time
: Total system run time for live threads.

RELATED INFORMATION

Functions: **task_info**.

Data Structures: **task_basic_info, thread_info**.

thread_basic_info

Structure — Defines basic information for threads

SYNOPSIS

```
struct thread_basic_info
{
        tvalspec_t          user_time;
        tvalspec_t          user_idle_time;
        tvalspec_t          system_time;
        cpu_usage_t         cpu_usage;
        policy_t            policy;
        run_state_t         run_state;
        thread_flags_t      flags;
        suspend_t           suspend_count;
        time_t              sleep_time;
};
typedef struct thread_basic_info*    thread_basic_info_t;
```

DESCRIPTION

The **thread_basic_info** structure defines the basic information array for threads. The **thread_info** function returns this array for a specified thread.

FIELDS

user_time
The total user run time for the thread.

user_idle_time
The total user run time with scheduling depressed for terminated threads within the task.

system_time
The total system run time for the thread.

cpu_usage
Scaled CPU usage percentage for the thread.

policy
Scheduling policy in effect

run_state
The thread's run state. Possible values are:

TH_STATE_RUNNING
The thread is running normally.

TH_STATE_STOPPED
The thread is stopped.

TH_STATE_WAITING
The thread is waiting normally.

TH_STATE_UNINTERRUPTIBLE
The thread is in an un-interruptible wait state.

TH_STATE_HALTED
The thread is halted at a clean point.

flags
Swap/idle flags for the thread. Possible values are:

TH_FLAGS_SWAPPED
The thread is swapped out.

TH_FLAGS_IDLE
The thread is an idle thread.

suspend_count
The current suspend count for the thread.

sleep_time
The number of seconds that the thread has been sleeping.

RELATED INFORMATION

Functions: **thread_info**.

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

tvalspec

Structure — Defines format of system time values

SYNOPSIS

[1]	struct **tvalspec**	
[2]	{	
[3]	unsigned_int	*tv_sec*;
[4]	clock_res_t	*tv_nsec*;
[5]	};	
[6]	typedef struct tvalspec	**tvalspec_t**;

DESCRIPTION

The **tvalspec** structure defines the format of the time structure supplied to or returned from the kernel. This definition conforms to the Posix 1003.4 *timespec* definition where the *tv_nsec* structure member is valid if ($0 \le tv\_nsec < 10^9$) and the time period described is $((tv\_sec \times 10^9) + tv\_nsec)$ nanoseconds.

FIELDS

tv_sec
: Seconds.

tv_nsec
: Nanoseconds.

RELATED INFORMATION

Functions: **clock_get_time, clock_set_time**, **clock_sleep**, **clock_alarm**, **alarm_reply.**

Data Structures: **mapped_tvalspec**.

vm_region_basic_info

Structure — Defines the attributes of a task's memory region.

SYNOPSIS

```
struct vm_region_basic_info
{
        vm_prot_t                       protection;
        vm_prot_t                       max_protection;
        vm_inherit_t                    inheritance;
        vm_offset_t                     offset;
        vm_behavior_t                   behavior;
        boolean_t                       wired;
};
typedef struct vm_region_basic_info*    vm_region_basic_info_t;
```

DESCRIPTION

The **vm_region_basic_info** structure defines the attributes of a memory region returned by **vm_region**.

FIELDS

protection
: The current protection for the region.

max_protection
: The maximum protection allowed for this region.

inheritance
: The inheritance attribute for the region.

offset
: The region's offset into the memory object. The region begins at this offset.

behavior
: Expected reference pattern for the memory. Valid values are:

VM_BEHAVIOR_DEFAULT
: The system's default behavior.

VM_BEHAVIOR_RANDOM
: No particular order expected.

VM_BEHAVIOR_SEQUENTIAL
: Forward sequential order.

VM_BEHAVIOR_RSEQUNTL
Reverse sequential order.

wired
True if the region is wired.

RELATED INFORMATION

Functions: **vm_region**, **vm_inherit**, **vm_reserve**, **vm_behavior_set**, **vm_protect**.

vm_statistics

Structure — Defines statistics for the kernel's use of virtual memory

SYNOPSIS

```
struct vm_statistics
{
        vm_size_t                       free_count;
        vm_size_t                       active_count;
        vm_size_t                       inactive_count;
        vm_size_t                       wire_count;
        vm_size_t                       zero_fill_count;
        vm_size_t                       reactivations;
        vm_size_t                       pageins;
        vm_size_t                       pageouts;
        vm_size_t                       faults;
        vm_size_t                       cow_faults;
        vm_size_t                       lookups;
        vm_size_t                       hits;
};
typedef struct vm_statistics*           vm_statistics_t;
```

DESCRIPTION

The **vm_statistics** structure defines the statistics available on the kernel's use of virtual memory. The statistics record virtual memory usage since the kernel was booted.

For related information for a specific task, see the **task_basic_info** structure.

FIELDS

free_count
: The total number of free pages in the system.

active_count
: The total number of pages currently in use and pageable.

inactive_count
: The number of inactive pages.

wire_count
: The number of pages that are wired in memory and cannot be paged out.

zero_fill_count
: The number of zero-fill pages.

reactivations
The number of reactivated pages.

pageins
The number of requests for pages from a pager (such as the i-node pager).

pageouts
The number of pages that have been paged out.

faults
The number of times the **vm_fault** routine has been called.

cow_faults
The number of copy-on-write faults.

lookups
The number of object cache lookups.

hits
The number of object cache hits.

NOTES

This structure is machine word length specific because of the memory sizes returned.

RELATED INFORMATION

Functions: **task_info**, **host_statistics**, **host_page_size**.

Data Structures: **task_basic_info**.

APPENDIX D Error Return Values

This appendix lists the various kernel return values.

Error Code Format

An error code has the following format:

- system code (6 bits). The **err_get_system** (*err*) macro extracts this field.
- subsystem code (12 bits). The **err_get_sub** (*err*) macro extracts this field.
- error code (14 bits). The **err_get_code** (*err*) macro extracts this field.

The various system codes are:

- *err_kern* —kernel
- *err_us* — user space library
- *err_server*— user space servers
- *err_mach_ipc* — IPC errors
- *err_local* — user defined errors

A typical user error code definition would be:

#define SOMETHING_WRONG *err_local* | **err_sub** (13) | 1

MIG Stub Errors

MIG_ARRAY_TOO_LARGE

User specified array not large enough to hold returned array.

MIG_BAD_ARGUMENTS

Message receiver found an invalid message size, invalid header fields or invalid descriptors. This could only happen if an invalidly formatted message (i.e., one that did not pass through Mach IPC) were passed to a MIG de-multiplexing routine.

MIG_BAD_ID

Bad message ID. This is only returned by MIG de-multiplexing routines when the message ID in the supplied message is not handled by that routine.

MIG_NO_REPLY

Not really an error, this is the code passed to MIG routines indicating that no reply is to be sent.

MIG_REPLY_MISMATCH

The message ID in a reply message does not correspond with the message ID of the request message.

MIG_SERVER_DIED

Message recipient no longer exists, or the recipient destroyed the request message without replying.

MIG_TYPE_ERROR

The wrong number or size of data or rights was received.

IPC Send Errors

The following errors can occur when **mach_msg** is used with the MACH_SEND_MSG option. This is also the case for all function interfaces.

MACH_SEND_INVALID_DATA

Message buffer is unreadable.

MACH_SEND_MSG_TOO_SMALL

Message buffer doesn't contain a complete message.

MACH_SEND_NO_BUFFER

A resource shortage prevented allocation of a message buffer.

IPC Receive Errors

The following errors can be returned by **mach_msg** when used with the MACH_RCV_MSG option. They can occur for kernel function interfaces.

MACH_RCV_IN_SET

The receive port name specified to **mach_msg** is a member of a port set.

MACH_RCV_INVALID_DATA

The message buffer was not writable.

MACH_RCV_INVALID_NAME

The receive port name specified to **mach_msg** is MACH_PORT_NULL, MACH_PORT_DEAD, names a null or dead right or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_RCV_LIMITS

The combined size of all out-of-line regions, the total of all port rights or the total message size exceeds the transfer limit for the port.

MACH_RCV_PORT_CHANGED

Receive right specified to **mach_msg** was moved into a set during the receive.

MACH_RCV_PORT_DIED

Receive right (or set) specified to **mach_msg** was sent away/died during receive.

MACH_RCV_TOO_LARGE

Message buffer is not large enough for the message.

Generic Kernel Errors

KERN_SUCCESS

Successful completion.

KERN_INVALID_ARGUMENT

The function requested was not applicable to this type of object.

KERN_INVALID_CAPABILITY

The supplied right is dead, null or not of the proper type.

KERN_INVALID_VALUE

A parameter's value was out of range (or possibly ill-formed). Specific error return values are returned if the parameter's value is properly formed and in range, but not a usable value at this time.

KERN_NO_SPACE

The task's port name space is full.

Port Manipulation Errors

KERN_INVALID_NAME

The port name doesn't denote a right in the task.

KERN_INVALID_RIGHT

The port name denotes a right, but not an appropriate right.

KERN_RESOURCE_SHORTAGE

A system resource could not be allocated to fulfill this request. This failure may be temporary.

Virtual Memory Manipulation Errors

KERN_INVALID_ADDRESS

Specified virtual address is not currently valid.

KERN_INVALID_OBJECT

The external memory manager failed to initialize the memory object.

KERN_MEMORY_ERROR

During a page fault, the memory object indicated that the data could not be returned. This failure may be temporary; future attempts to access this same data may succeed, as defined by the memory object.

KERN_PROTECTION_FAILURE

Specified memory is valid, but does not permit the required forms of access or the protection being requested exceeds that permitted.

KERN_RESOURCE_SHORTAGE

A system resource could not be allocated to fulfill this request. This failure may be temporary.

Random Kernel Errors

KERN_ABORTED

The operation was aborted.

KERN_FAILURE

A catch-all error for implementation dependent failures.

KERN_INVALID_CLOCK

The port was not a clock port.

KERN_INVALID_DEVICE

The port was not a device port.

KERN_INVALID_HOST

An argument supplied to assert system privilege was not a host control port.

KERN_INVALID_LEDGER

A ledger was required but not supplied.

KERN_INVALID_MEMORY_CONTROL

The port was not a memory cache control port.

KERN_INVALID_POLICY

The specified scheduling policy is not currently enabled for the processor set.

KERN_INVALID_PROCESSOR_SET

An argument supplied to assert processor set privilege was not a processor set control port.

KERN_INVALID_SECURITY

An argument supplied to assert security privilege was not a host security port.

KERN_INVALID_TASK

A task was required but not supplied.

KERN_POLICY_LIMIT

The specified scheduling attributes exceeds the thread's limits.

KERN_INVALID_THREAD

A thread was required but not supplied.

Kernel Device Errors

D_SUCCESS

Normal device return. This is the same value as KERN_SUCCESS.

D_ALREADY_OPEN

Exclusive-use device already open.

D_DEVICE_DOWN

Device has been shut down.

D_INVALID_OPERATION

The requested operation or option is invalid for this device.

D_INVALID_RECNUM

Invalid record (block) number.

D_INVALID_SIZE

Invalid I/O size, including request for inband transfer exceeding the limit for inband data.

D_IO_ERROR

Hardware I/O error.

D_NO_SUCH_DEVICE

No device with that name, security ID mismatch, 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.

APPENDIX E

Object Index

APPENDIX F

Interface and Structure Index