Mach 3 Kernel Interfaces

Open Software Foundation and Carnegie Mellon University

Keith Loepere, Editor

OSF

NORMA-MK12: July 15, 1992

This book is in the Open Software Foundation Mach 3 series.

Books in the OSF Mach 3 series:

Mach 3 Kernel Principles

Mach 3 Kernel Interfaces

Mach 3 Server Writer's Guide

Mach 3 Server Writer's Interfaces

Revision History:

Revision 2	MK67: January 7, 1992	OSF / Mach release
Revision 2.2	NORMA-MK12: July 15, 1992	

Change bars indicate changes since MK67.

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. and Michael W. Young.

Contents

CHAPTER 1 Introduction

This book documents the various interfaces to the Mach 3 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 a companion volume.

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 difficult and highly subjective process. For example, an interface that returns the list of processor sets defined for a host can be grouped with host related interfaces or processor set related interfaces. 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 and an alphabetical index of functions and data structures.

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

- The pertinent library. All functions in this volume are contained in **libmach_sa.a** (and, by implication, **libmach.a**) unless otherwise noted. Also listed is the header file that provides the function prototype or defines the data structure (if not **mach.h**).
- A synopsis of the interface, in C form
- An extended description of the function performed by the call
- Any macro or special forms of 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

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 invoke 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 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 the Net Message server to work.

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. (Some functions are a hybrid; they first try the system call, and, failing that, they try sending a Mach message. This is an optimization for some interfaces for which the target is usually the invoking 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.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 (probably with **mach_msg_server**) 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**) which 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 header 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 which 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.

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. Note that port types are of this form.

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

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.

An “unbounded out in-line array” specifies the variable in-line/out-of-line (referred to as unbounded in-line) array feature of MIG described in the *Server Writer’s Guide*. The caller supplies a pointer to a pointer whose value contains the address of an array whose size is specified in some other parameter (or known implicitly). Upon return, if this target pointer no longer points to the caller’s array (most likely because the caller’s array was not sufficiently large to hold the return data), then the kernel allocated space (as if by **vm_allocate**) into which the data was placed; otherwise, the data was placed into the supplied array.

CHAPTER 2 IPC Interface

This chapter discusses the specifics of the kernel's inter-"process" communication (IPC) interfaces. The interfaces discussed are only the interfaces directly involved in sending and receiving IPC messages.

mach_msg

System Trap / Function — Sends and receives a message using the same message buffer

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_timeout_t         timeout,
        mach_port_t                notify);
```

DESCRIPTION

The **mach_msg** system call sends and receives Mach messages. Mach messages contain typed data, which can include port rights and addresses of large regions of memory.

If the *option* argument contains MACH_SEND_MSG, it sends a message. The *send_size* argument specifies the size of the message 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 the message buffer that will receive the message; messages larger than *rcv_size* 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. 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.

Some options, like MACH_SEND_TIMEOUT and MACH_RCV_TIMEOUT, share a supporting argument. If these options are used together, they make independent use of the supporting argument's value.

PARAMETERS

msg
: [pointer to in/out structure] A message buffer. This should be aligned on a long-word boundary.

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. Otherwise zero should be supplied.

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

rcv_name
: [in scalar] When receiving a message, specifies the port or port set. Otherwise MACH_PORT_NULL should be supplied.

timeout
: [in scalar] When using the MACH_SEND_TIMEOUT and MACH_-RCV_TIMEOUT options, specifies the time in milliseconds to wait before giving up. Otherwise MACH_MSG_TIMEOUT_NONE should be supplied.

notify
: [in scalar] When using the MACH_SEND_NOTIFY, MACH_SEND_-CANCEL, and MACH_RCV_NOTIFY options, specifies the port used for the notification. Otherwise MACH_PORT_NULL should be supplied.

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. Messages contain a fixed-size header and a variable amount of typed data following the header. The header describes the destination and size of the message.

The IPC implementation makes use of the VM system to efficiently transfer large amounts of data. The message body can contain an address of a region of the sender's address space which should be transferred as part of the message. When a task receives a message containing an out-of-line region of data, the data appears in an unused portion of the receiver's address space. This transmission of out-of-line data is optimized so that sender and receiver share the physical pages of data copy-on-write, and no actual data copy occurs unless the pages are written. Regions of memory up to the size of a full address space may be sent in this manner.

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 body 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 integers. Except for the reserved values MACH_PORT_NULL (0) and MACH_-

PORT_DEAD (-1), this is a full 32-bit name space. When the kernel chooses a name for a new right, it is free to pick any unused name (one which denotes no right) in the space.

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, a port-set, be a dead name, or name nothing. Dead names are not capabilities. They act as place-holders to prevent a name from being otherwise used.

A port is destroyed, or dies, when its receive right is de-allocated. When a port dies, send and send-once rights for the port turn into dead names. Any messages queued at the port are destroyed, which de-allocates the port rights and out-of-line memory in the messages.

Tasks may hold multiple user-references for send rights and dead names. When a task receives a send right which 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.

Send-once rights always have a user-reference count of one, although a port can have multiple send-once rights, because each send-once right held by a task has a different name. In contrast, when a task holds send rights or a receive right for a port, the rights share a single name.

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 message body can carry port rights; the *msgt_name* (*msgtl_name*) field in a type descriptor specifies the type of port right and how the port right is to be extracted from the caller. The values MACH_PORT_NULL and MACH_PORT_DEAD are always valid in place of a port right in a message body.

In a sent message, the following *msgt_name* 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, and the receive right's make-send count is incremented.

MACH_MSG_TYPE_COPY_SEND
: The message will carry a send right, and the caller should supply a send right. The user reference count for the supplied send right is not changed. The caller may also supply a dead name and the receiving task will get MACH_PORT_DEAD.

MACH_MSG_TYPE_MOVE_SEND
: The message will carry a send right, and the caller should 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. The caller may also supply a dead name, which loses a user reference, and the receiving task will get MACH_PORT_DEAD.

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 should supply a send-once right. The caller loses the supplied send-once right. The caller may also supply a dead name, which loses a user reference, and the receiving task will get MACH_PORT_DEAD.

MACH_MSG_TYPE_MOVE_RECEIVE
: The message will carry a receive right, and the caller should supply a receive right. The caller loses the supplied receive right, but retains any send rights with the same name.

If a message carries a send or send-once right, and the port dies while the message is in transit, then the receiving task will get MACH_PORT_DEAD instead of a right.

The following *msgt_name* 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 new 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. Oth-

erwise, the right will have a new, previously unused, name. The make-send count and sequence number of the receive right are reset to zero, but the port retains other attributes like queued messages, extant send and send-once rights, and requests for port-destroyed and no-senders notifications. (Note: It is currently planned to remove port-destroyed notifications from the kernel interface and to define no-senders notifications as being canceled when a receive right is moved.)

Memory

A message body can contain an address of a region of the sender's address space which should be transferred as part of the message. The message carries a logical copy of the memory, but the kernel uses VM techniques to defer any actual page copies. Unless the sender or the receiver modifies the data, the physical pages remain shared.

An out-of-line transfer occurs when the data's type descriptor specifies *msgt_inline* as FALSE. The address of the memory region should follow the type descriptor in the message body. The type descriptor and the address contribute to the message's size (*send_size*, *msgh_size*). The out-of-line data does not contribute to the message's size.

The name, size, and number fields in the type descriptor describe the type and length of the out-of-line data, not the address. Out-of-line memory frequently requires long type descriptors (**mach_msg_type_long_t**), because the *msgt_number* field is too small to describe a page of 4K bytes.

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**'ed memory. The receiver has the responsibility of de-allocating (with **vm_deallocate**) the memory when it is no longer needed. Security-conscious receivers should exercise caution when dealing with out-of-line memory from un-trustworthy sources, because the memory may be backed by an unreliable memory manager.

Null out-of-line memory is legal. If the out-of-line region size is zero (for example, because *msgtl_number* is zero), then the region's specified address is ignored. A received 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 funny sizes. In the general case, the first and last pages in the new memory region in the receiver do not contain data from the sender, but are partly zero. The received address points into the middle of the first page. This possibility doesn't complicate de-allocation, because **vm_deallocate** does the right thing, rounding the start address down and the end address up to de-allocate all arrived pages.

Out-of-line memory has a de-allocate option, controlled by the *msgt_deallocate* bit. If it is TRUE 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 and end addresses are rounded so that every page overlapped by the memory region is de-allocated. The use of *msgt_deallocate* effectively changes the memory copy into a memory movement. In a received message, *msgt_deallocate* is TRUE in type descriptors for out-of-line memory.

Out-of-line memory can carry port rights.

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.

Message delivery is reliable and sequenced. Messages are not lost, and messages sent to a port from a single thread are received in the order in which they were sent.

If the destination port's queue is full, then several things can happen. If the message is sent to a send-once right (*msgh_remote_port* carries a send-once right), then the kernel ignores the queue limit and delivers the message. Otherwise the caller blocks until there is room in the queue, unless the MACH_SEND_TIMEOUT or MACH_SEND_NOTIFY options are used. If a port has several blocked senders, then any of them may queue the next message when space in the queue becomes available, with the proviso that a blocked sender will not be indefinitely starved.

These options modify MACH_SEND_MSG. If MACH_SEND_MSG is not also specified, they are ignored.

MACH_SEND_TIMEOUT

The *timeout* argument should specify a maximum time (in milliseconds) for the call to block before giving up. If the message can't be queued before the timeout interval elapses, then the call returns MACH_SEND_TIMED_OUT. A zero timeout is legitimate.

MACH_SEND_NOTIFY

The *notify* argument should specify a receive right for a notify port. If the send were to block, then instead the message is queued, MACH_SEND_WILL_NOTIFY is returned, and a msg-accepted notification is requested. If MACH_SEND_TIMEOUT is also specified, then MACH_SEND_NOTIFY doesn't take effect until the timeout interval elapses.

Only one message at a time can be forcibly queued to a send right with MACH_SEND_NOTIFY. A msg-accepted notification is sent to the notify port when another message can be forcibly queued. If an attempt is made to use MACH_SEND_NOTIFY before then, the call returns a MACH_SEND_NOTIFY_IN_PROGRESS error.

The msg-accepted notification carries the name of the send right. If the send right is de-allocated before the msg-accepted notification is generated, then the msg-accepted notification carries the value MACH_PORT_NULL. If the destination port is destroyed before the notification is generated, then a send-once notification is generated instead.

(Note: It is currently planned that this option will be deleted, as well as the provision of the corresponding notification.)

MACH_SEND_INTERRUPT
: If specified, the **mach_msg** call will return MACH_SEND_INTERRUPTED if a software interrupt aborts the call. Otherwise, the send operation will be retried.

MACH_SEND_CANCEL
: The *notify* argument should specify a receive right for a notify port. If the send operation removes the destination port right from the caller, and the removed right had a dead-name request registered for it, and *notify* is the notify port for the dead-name request, then the dead-name request may be silently canceled (instead of resulting in what would have been a port-deleted notification).

  This option is typically used to cancel a dead-name request made with the MACH_RCV_NOTIFY option. It should only be used as an optimization.

Some return codes, like MACH_SEND_TIMED_OUT, imply that the message was almost sent, but could not be queued. In these situations, the kernel tries to return the message contents to the caller with a pseudo-receive operation. This prevents the loss of port rights or memory which only exist in the message. For example, a receive right which was moved into the message, or out-of-line memory sent with the de-allocate bit.

The pseudo-receive operation is very similar to a normal receive operation. The pseudo-receive handles the port rights in the message header as if they were in the message body. They are not reversed (as is the appearance in a normal received message). After the pseudo-receive, the message is ready to be resent. If the message is not resent, note that out-of-line memory regions may have moved and some port rights may have changed names.

The pseudo-receive operation may encounter resource shortages. This is similar to a MACH_RCV_BODY_ERROR return code from a receive operation. When this happens, the normal send return codes are augmented with the MACH_MSG_IPC_SPACE, MACH_MSG_VM_SPACE, MACH_MSG_IPC_KERNEL, and MACH_MSG_VM_KERNEL bits to indicate the nature of the resource shortage.

The queueing of a message carrying receive rights may create a circular loop of receive rights and messages, which can never be received. For example, a message carrying a receive right can be sent to that receive right. This situation is not an error, but the kernel will garbage-collect such loops, destroying the messages.

Message Receive

The receive operation de-queues 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, subject to the MACH_RCV_TIMEOUT option.

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 which 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 specifies from which port in the port set the message came.

The *rcv_size* argument specifies the size of the caller's message buffer. The **mach_msg** call will not receive a message larger than *rcv_size*. Messages that are too large are destroyed, unless the MACH_RCV_LARGE option is used.

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 bits in *msgh_bits* are also reversed. The MACH_MSGH_BITS_LOCAL bits have the value MACH_MSG_TYPE_PORT_SEND if the message was sent to a send right, and the value MACH_MSG_TYPE_PORT_SEND_ONCE if was sent to a send-once right. The MACH_MSGH_BITS_REMOTE bits describe the reply port right.

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, and the sequence number is reset to zero whenever the port's receive right moves between tasks. When a message is de-queued from the port, it is stamped with the port's sequence number and the port's sequence number is then incremented. The de-queue and increment operations are atomic, so that multiple threads receiving messages from a port can use the *msgh_seqno* field to reconstruct the original order of the messages.

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 destroys the send or send-once right for the destination port. The *msgh_remote_*-

port field does carry a port right, and the message body can carry port rights and memory if MACH_MSGH_BITS_COMPLEX is present in *msgh_bits*. 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*. If other threads are concurrently manipulating the receive right, the situation is more complicated. If the receive right is renamed during the call, then *msgh_local_port* specifies the right's new name. If the caller loses the receive right after the message was de-queued from it, then **mach_msg** will proceed instead of returning MACH_RCV_PORT_DIED. If the receive right was destroyed, then *msgh_local_port* specifies MACH_PORT_DEAD. If the receive right still exists, but isn't held by the caller, then *msgh_local_port* specifies MACH_PORT_NULL.

These options modify MACH_RCV_MSG. If MACH_RCV_MSG is not also specified, they are ignored.

MACH_RCV_TIMEOUT
: The *timeout* argument should specify a maximum time (in milliseconds) for the call to block before giving up. If no message arrives before the timeout interval elapses, then the call returns MACH_RCV_TIMED_OUT. A zero timeout is legitimate.

MACH_RCV_NOTIFY
: The *notify* argument should specify a receive right for a notify port. If receiving the reply port creates a new port right in the caller, then the notify port is used to request a dead-name notification for the new port right.

MACH_RCV_INTERRUPT
: If specified, the **mach_msg** call will return MACH_RCV_INTERRUPTED if a software interrupt aborts the call. Otherwise, the receive operation will be retried.

MACH_RCV_LARGE
: If the message is larger than *rcv_size*, then the message remains queued instead of being destroyed. The call returns MACH_RCV_TOO_LARGE and the actual size of the message is returned in the *msgh_size* field of the message header. If this option is not specified, messages too large will be de-queued and then destroyed; the caller receives the message's header, with all fields correct, including the destination port but excepting the reply port, which is MACH_PORT_NULL.

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. In addition, the *msgt_size* (*msgtl_-size*) field in the region's type descriptor is changed to zero. If a resource shortage prevents the reception of out-of-line memory carrying port rights, then the port rights are always destroyed if the memory region can not be received. A task never receives port rights or memory for which it is not told.

The MACH_RCV_HEADER_ERROR return code indicates a resource shortage in the reception of the message's header. The reply port and all port rights and memory in the message body are destroyed. The caller receives the message's header, with all fields correct except for the reply port.

The MACH_RCV_BODY_ERROR return code indicates a resource shortage in the reception of the message's body. The message header, including the reply port, is correct. The kernel attempts to transfer all port rights and memory regions in the body, and only destroys those that can't be transferred.

Atomicity

The **mach_msg** call handles port rights in a message header atomically. Port rights and out-of-line memory in a message body do not enjoy this atomicity guarantee. The message body may be processed front-to-back, back-to-front, first out-of-line memory then port rights, in some random order, or even atomically.

For example, consider sending a message with the destination port specified as MACH_MSG_TYPE_MOVE_SEND and the reply port specified as MACH_MSG_TYPE_COPY_SEND. The same send right, with one user-reference, is supplied for both the *msgh_remote_port* and *msgh_local_port* fields. Because **mach_msg** processes the message header atomically, this succeeds. If *msgh_remote_port* were processed before *msgh_local_port*, then **mach_msg** would return MACH_SEND_INVALID_REPLY in this situation.

On the other hand, suppose the destination and reply port are both specified as MACH_MSG_TYPE_MOVE_SEND, and again the same send right with one user-reference is supplied for both. Now the send operation fails, but because it processes the header atomically, **mach_msg** can return either MACH_-SEND_INVALID_DEST or MACH_SEND_INVALID_REPLY.

For example, consider receiving a message at the same time another thread is de-allocating the destination receive right. Suppose the reply port field carries a send right for the destination port. If the de-allocation happens before the de-queuing, then the receiver gets MACH_RCV_PORT_DIED. If the de-allocation happens after the receive, then the *msgh_local_port* and the *msgh_remote_port* fields both specify the same right, which becomes a dead name when the receive right is de-allocated. If the de-allocation happens between the de-queue and the receive, then the *msgh_local_port* and *msgh_remote_port* fields both specify MACH_PORT_DEAD. Because the header is processed atomically, it is not possible for just one of the two fields to hold MACH_PORT_DEAD.

The MACH_RCV_NOTIFY option provides a more likely example. Suppose a message carrying a send-once right reply port is received with MACH_-RCV_NOTIFY at the same time the reply port is destroyed. If the reply port is destroyed first, then *msgh_remote_port* specifies MACH_PORT_DEAD and the kernel does not generate a dead-name notification. If the reply port is destroyed after it is received, then *msgh_remote_port* specifies a dead name for which the kernel generates a dead-name notification. It is not possible to receive the reply port right and have it turn into a dead name before the dead-name notification is requested; as part of the message header the reply port is received atomically.

Implementation

mach_msg is a wrapper for a system call. **mach_msg** has the responsibility for repeating the interrupted system call.

CAUTIONS

Sending out-of-line memory with a non-page-aligned address, or a size which is not a page multiple, works but with a caveat. The extra bytes in the first and last page of the received memory are not zeroed, so the receiver can peek at more data than the sender intended to transfer. This might be a security problem for the sender.

If MACH_RCV_TIMEOUT is used without MACH_RCV_INTERRUPT, then the timeout duration might not be accurate. When the call is interrupted and automatically retried, the original timeout is used. If interrupts occur frequently enough, the timeout interval might never expire. MACH_SEND_TIMEOUT without MACH_SEND_INTERRUPT suffers from the same problem.

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_NO_BUFFER
: A resource shortage prevented the kernel from allocating a message buffer.

MACH_SEND_INVALID_DATA
: The supplied message buffer was not readable.

MACH_SEND_INVALID_HEADER
: The *msgh_bits* value was invalid.

MACH_SEND_INVALID_DEST
: The *msgh_remote_port* value was invalid.

MACH_SEND_INVALID_REPLY
: The *msgh_local_port* value was invalid.

MACH_SEND_INVALID_NOTIFY
: When using MACH_SEND_CANCEL, the *notify* argument did not denote a valid receive right.

These return codes imply that some or all of the message was destroyed:

MACH_SEND_INVALID_MEMORY
: The message body specified out-of-line data that was not readable.

MACH_SEND_INVALID_RIGHT
: The message body specified a port right which the caller didn't possess.

MACH_SEND_INVALID_TYPE
: A type descriptor was invalid.

MACH_SEND_MSG_TOO_SMALL
: The last data item in the message ran over the end of the message.

These return codes imply that the message was returned to the caller with a pseudo-receive operation:

MACH_SEND_TIMED_OUT
: The *timeout* interval expired.

MACH_SEND_INTERRUPTED
: A software interrupt occurred.

MACH_SEND_INVALID_NOTIFY
: When using MACH_SEND_NOTIFY, the *notify* argument did not denote a valid receive right.

MACH_SEND_NO_NOTIFY
: A resource shortage prevented the kernel from setting up a msg-accepted notification.

MACH_SEND_NOTIFY_IN_PROGRESS
: A msg-accepted notification was already requested, and hasn't yet been generated.

These return codes imply that the message was queued:

MACH_SEND_WILL_NOTIFY
: The message was forcibly queued, and a msg-accepted notification was requested.

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 de-queue a message:

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_TIMED_OUT
: The *timeout* interval expired.

MACH_RCV_INTERRUPTED
: A software interrupt occurred.

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

MACH_RCV_PORT_CHANGED
: *rcv_name* specified a receive right which was moved into a port set during the call.

MACH_RCV_TOO_LARGE
: When using MACH_RCV_LARGE, and the message was larger than *rcv_size*. The message is left queued, and its actual size is returned in the *msgh_size* field of the message buffer.

These return codes imply that a message was de-queued and destroyed:

MACH_RCV_HEADER_ERROR
: A resource shortage prevented the reception of the port rights in the message header.

MACH_RCV_INVALID_NOTIFY
: When using MACH_RCV_NOTIFY, the *notify* argument did not denote a valid receive right.

MACH_RCV_TOO_LARGE
: When not using MACH_RCV_LARGE, a message larger than *rcv_size* was de-queued and destroyed.

These return codes imply that a message was received:

MACH_RCV_BODY_ERROR
: A resource shortage prevented the reception of a port right or out-of-line memory region in the message body.

MACH_RCV_INVALID_DATA
: The specified message buffer was not writable. The calling task did successfully receive the port rights and out-of-line memory regions in the message.

MACH_MSG_SUCCESS
: A message was received.

Resource shortages can occur after a message is de-queued, while transferring port rights and out-of-line memory regions to the receiving task. The **mach_msg** call returns MACH_RCV_HEADER_ERROR or MACH_RCV_-BODY_ERROR in this situation. These return codes always carry extra bits (bit-wise-or'ed) that indicate the nature of the resource shortage:

MACH_MSG_IPC_SPACE
: There was no room in the task's IPC name space for another port name.

MACH_MSG_VM_SPACE
: There was no room in the task's VM address space for an out-of-line memory region.

MACH_MSG_IPC_KERNEL
: A kernel resource shortage prevented the reception of a port right.

MACH_MSG_VM_KERNEL
: A kernel resource shortage prevented the reception of an out-of-line memory region.

RELATED INFORMATION

Functions: **mach_msg_receive**, **mach_msg_send**.

Data Structures: **mach_msg_header**, **mach_msg_type**, **mach_msg_type_long, mach_msg_accepted_notification**, **mach_send_once_notification**.

mach_msg_receive

Function — Receives a message from a port or port set

LIBRARY

Not declared anywhere.

SYNOPSIS

mach_msg_return_t **mach_msg_receive**
(mach_msg_header_t* *header)*;

DESCRIPTION

The **mach_msg_receive** function is a shorthand for the following call:

mach_msg (*header*, MACH_RCV_MSG, 0, *header→msgh_size*,
header→msgh_local_port, MACH_MSG_TIMEOUT_NONE,
MACH_PORT_NULL);

PARAMETERS

header
[pointer to in/out structure] The address of the buffer that is to receive the message. The *msgh_local_port* and *msgh_size* fields in *header* must be set.

RETURN VALUE

Refer to **mach_msg** for a description of the various receive errors.

RELATED INFORMATION

Functions: **mach_msg**, **mach_msg_send**.

Data Structures: **mach_msg_header**, **mach_msg_type**, **mach_msg_type_long**.

mach_msg_send

Function — Sends a message to a port

LIBRARY

Not declared anywhere.

SYNOPSIS

mach_msg_return_t **mach_msg_send**
(mach_msg_header_t* *header*);

DESCRIPTION

The **mach_msg_send** function is a shorthand for the following call:

mach_msg (*header*, MACH_SEND_MSG, *header*→*msgh_size*, 0,
MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE,
MACH_PORT_NULL);

PARAMETERS

header
: [pointer to in structure] The address of the buffer that contains the message to be sent.

RETURN VALUE

Refer to **mach_msg** for a description of the send errors.

RELATED INFORMATION

Functions: **mach_msg**, **mach_msg_receive**.

Data Structures: **mach_msg_header**, **mach_msg_type**, **mach_msg_type_long**.

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.

do_mach_notify_dead_name

Server Interface — Handles the occurrence of a dead-name notification

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **do_mach_notify_dead_name**
(notify_port_t *notify,*
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**.

SEQUENCE NUMBER FORM

do_seqnos_mach_notify_dead_name
kern_return_t **do_seqnos_mach_notify_dead_name**
(notify_port_t *notify,*
mach_port_seqno_t *seqno*,
mach_port_name_t *name)*;

PARAMETERS

notify
[in scalar] 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.

RETURN VALUE

KERN_SUCCESS
The notification was received.

RELATED INFORMATION

Functions: **notify_server**, **mach_msg**, **mach_port_request_notification,** **do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_port_deleted**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once.**

do_mach_notify_msg_accepted

Server Interface — Handles the occurrence of a message accepted notification

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t do_mach_notify_msg_accepted
                (notify_port_t                notify,
                 mach_port_name_t             name);
```

DESCRIPTION

A **do_mach_notify_msg_accepted** function is called by **notify_server** as the result of a kernel message indicating that a message forcibly queued to a port via MACH_NOTIFY_SEND was accepted. *notify* is the port named via **mach_msg**.

(Note: This feature is current planned for deletion.)

SEQUENCE NUMBER FORM

do_seqnos_mach_notify_msg_accepted

```
kern_return_t do_seqnos_mach_notify_msg_accepted
                (notify_port_t                notify,
                 mach_port_seqno_t            seqno,
                 mach_port_name_t             name);
```

PARAMETERS

notify
: [in scalar] 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 port whose message was accepted.

RETURN VALUE

KERN_SUCCESS
: The notification was received.

RELATED INFORMATION

Functions: **notify_server**, **mach_msg**, **mach_port_request_notification, do_mach_notify_dead_name**, **do_mach_notify_no_senders**, **do_mach_notify_port_deleted**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once.**

do_mach_notify_no_senders

Server Interface — Handles the occurrence of a no-more-senders notification

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t do_mach_notify_no_senders
        (notify_port_t              notify,
         mach_port_mscount_t        mscount);
```

DESCRIPTION

A **do_mach_notify_no_senders** function is called by **notify_server** as the result of a kernel message indicating that a receive right has no more senders. *notify* is the port named via **mach_port_request_notification**.

SEQUENCE NUMBER FORM

do_seqnos_mach_notify_no_senders

```
kern_return_t do_seqnos_mach_notify_no_senders
        (notify_port_t              notify,
         mach_port_seqno_t          seqno,
         mach_port_mscount_t        mscount);
```

PARAMETERS

notify
: [in scalar] The port to which the notification was sent.

seqno
: [in scalar] The sequence number of this message relative to the notification port.

mscount
: [in scalar] The value the port's make-send count had when it was generated.

RETURN VALUE

KERN_SUCCESS
: The notification was received.

RELATED INFORMATION

Functions: **notify_server**, **mach_msg**, **mach_port_request_notification, do_mach_notify_msg_accepted**, **do_mach_notify_dead_name**, **do_mach_notify_port_deleted**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once.**

do_mach_notify_port_deleted

Server Interface — Handles the occurrence of a port-deleted notification

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t do_mach_notify_port_deleted
        (notify_port_t                    notify,
         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**.

SEQUENCE NUMBER FORM

do_seqnos_mach_notify_port_deleted

```
kern_return_t do_seqnos_mach_notify_port_deleted
        (notify_port_t                    notify,
         mach_port_seqno_t                seqno,
         mach_port_name_t                 name);
```

PARAMETERS

notify
: [in scalar] 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.

RETURN VALUE

KERN_SUCCESS
: The notification was received.

RELATED INFORMATION

Functions: **notify_server**, **mach_msg**, **mach_port_request_notification, do_mach_notify_dead_name**, **do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once.**

do_mach_notify_port_destroyed

Server Interface — Handles the occurrence of a port destroyed notification

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t do_mach_notify_port_destroyed
                (notify_port_t                        notify,
                 mach_port_receive_t                  rights);
```

DESCRIPTION

A **do_mach_notify_port_destroyed** function is called by **notify_server** as the result of a kernel message indicating that a receive right would have been destroyed. *notify* is the port named via **mach_port_request_notification**.

(Note: This feature is currently planned for deletion.)

SEQUENCE NUMBER FORM

do_seqnos_mach_notify_port_destroyed

```
kern_return_t do_seqnos_mach_notify_port_destroyed
                (notify_port_t                        notify,
                 mach_port_seqno_t                    seqno,
                 mach_port_receive_t                  rights);
```

PARAMETERS

notify
: [in scalar] The port to which the notification was sent.

seqno
: [in scalar] The sequence number of this message relative to the notification port.

rights
: [in scalar] The receive right that would have been destroyed.

RETURN VALUE

KERN_SUCCESS
: The notification was received.

RELATED INFORMATION

Functions: **notify_server**, **mach_msg**, **mach_port_request_notification, do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_dead_name**, **do_mach_notify_port_deleted**, **do_mach_notify_send_once.**

do_mach_notify_send_once

Server Interface — Handles the occurrence of a send-once notification

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **do_mach_notify_send_once**
(notify_port_t *notify)*;

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 named via **mach_msg**.

SEQUENCE NUMBER FORM

do_seqnos_mach_notify_send_once
kern_return_t **do_seqnos_mach_notify_send_once**
(notify_port_t *notify,*
mach_port_seqno_t *seqno)*;

PARAMETERS

notify
[in scalar] The port to which the notification was sent.

seqno
[in scalar] The sequence number of this message relative to the notification port.

RETURN VALUE

KERN_SUCCESS
The notification was received.

RELATED INFORMATION

Functions: **notify_server**, **mach_msg**, **mach_port_request_notification, do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_port_deleted**, **do_mach_notify_port_destroyed**, **do_mach_notify_dead_name.**

mach_port_allocate

Function — Creates a port right

SYNOPSIS

kern_return_t **mach_port_allocate**

(mach_port_t	*task,*
mach_port_right_t	*right,*
mach_port_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

task

[in scalar] 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 make-send count is zero, its sequence number is zero, its queue limit is MACH_PORT_QLIMIT_DEFAULT, and it has no queued messages. *name* denotes the receive right for the new port.

task 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 dead name. The new dead name has one user reference.

name

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

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_VALUE
right was invalid.

KERN_NO_SPACE
There was no room in *task*'s IPC name space for another right.

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_allocate_name**, **mach_port_deallocate, mach_port_-insert_right, mach_port_extract_right**.

mach_port_allocate_name

Function — Creates a port right with a given name

SYNOPSIS

kern_return_t **mach_port_allocate_name**

(mach_port_t	*task,*
mach_port_right_t	*right*,
mach_port_t	*name*);

DESCRIPTION

The **mach_port_allocate_name** function creates a new right in the specified task, with a specified name for the new right.

PARAMETERS

task
: [in scalar] The task acquiring the port right.

right
: [in scalar] The kind of right which will be created. This is one of the following values:

MACH_PORT_RIGHT_RECEIVE
: **mach_port_allocate_name** 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 make-send count is zero, its sequence number is zero, its queue limit is MACH_PORT_QLIMIT_DEFAULT, and it has no queued messages. *name* denotes the receive right for the new port.

: *task* 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_name** creates a port set. The new port set has no members.

MACH_PORT_RIGHT_DEAD_NAME
: **mach_port_allocate_name** creates a new dead name. The new dead name has one user reference.

name
: [in scalar] The task's name for the port right. *name* must not already be in use for some right, and it can't be the reserved values MACH_-PORT_NULL and MACH_PORT_DEAD.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_VALUE
: *right* was invalid.

KERN_INVALID_VALUE
: *name* was MACH_PORT_NULL or MACH_PORT_DEAD.

KERN_NAME_EXISTS
: *name* was already in use for a port right.

KERN_RESOURCE_SHORTAGE
: The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_deallocate**, **mach_port_rename**.

mach_port_deallocate

Function — Releases a user reference for a right

SYNOPSIS

kern_return_t **mach_port_deallocate**
(mach_port_t *task,*
mach_port_t *name);*

DESCRIPTION

The **mach_port_deallocate** function releases a user reference for a right. It is an alternate form of **mach_port_mod_refs** that allows a task to release a user reference for a send or send-once right without failing if the port has died and the right is now actually a dead name.

If *name* denotes a dead name, send right, or send-once right, then the right loses one user reference. If it only had one user reference, then the right is destroyed.

PARAMETERS

task
[in scalar] The task holding the right.

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

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted an invalid right.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_allocate_name, mach_port_mod_refs**.

mach_port_destroy

Function — Removes a task's rights for a name

SYNOPSIS

```
kern_return_t mach_port_destroy
        (mach_port_t                    task;
         mach_port_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** and **mach_port_deallocate** are 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 there is a port-destroyed request registered for the port, then the receive right is not actually destroyed, but instead is sent in a port-destroyed notification. (Note: Port destroyed notifications are currently planned for deletion.) If there is no registered port-destroyed request, remaining messages queued to the port are destroyed and extant send and send-once rights turn into dead names. If those send and send-once rights have dead-name requests registered, then dead-name notifications are generated for them.

If *name* denotes a send-once right, then the send-once right is used to produce 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 the registered send-once right is used to produce a port-deleted notification for the name.

PARAMETERS

task
: [in scalar] The task holding the right.

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

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_allocate_name, mach_port_mod_refs**, **mach_port_deallocate**, **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 *task,*
mach_port_t *name,*
mach_msg_type_name_t *desired_type,*
mach_port_t* *right,*
mach_msg_type_name_t* *acquired_type)*;

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 *desired_type* as the value of *msgt_name*. See **mach_msg**.

The returned value of *acquired_type* 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

task
[in scalar] The task holding the port right.

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

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

right
[out scalar] The extracted right.

acquired_type
[out scalar] The type of the extracted right.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted an invalid right.

KERN_INVALID_VALUE
desired_type was invalid.

RELATED INFORMATION

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

mach_port_get_receive_status

Function — Returns the status of a receive right

SYNOPSIS

```
kern_return_t mach_port_get_receive_status
                (mach_port_t              task,
                 mach_port_t              name,
                 mach_port_status_t*      status);
```

DESCRIPTION

The **mach_port_get_receive_status** function returns the current status of the specified receive right.

PARAMETERS

task
: [in scalar] The task holding the receive right.

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

status
: [out structure] The status information for the receive right.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

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_set_qlimit**, **mach_port_set_mscount**, **mach_port_set_seqno**.

Data Structures: **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                task,
                 mach_port_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.

PARAMETERS

task
: [in scalar] The task holding the right.

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

right
: [in scalar] The type of right / entity 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.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_VALUE
right was invalid.

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_mod_refs**.

mach_port_get_set_status

Function — Returns the members of a port set

SYNOPSIS

kern_return_t **mach_port_get_set_status**

(mach_port_t	*task,*
mach_port_t	*name,*
mach_port_array_t*	*members,*
mach_msg_type_number_t*	*count)*;

DESCRIPTION

The **mach_port_get_set_status** function returns the members of a port set. *members* is an array that is automatically allocated when the reply message is received.

PARAMETERS

task
: [in scalar] The task holding the port set.

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

members
: [out pointer to dynamic array of *mach_port_t*] The task's names for the port set's members.

count
: [out scalar] The number of member names returned.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
: *name* did not denote a right.

KERN_INVALID_RIGHT
: *name* denoted a right, but not a port set.

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_move_member**, **vm_deallocate**.

mach_port_insert_right

Function — Inserts a port right into a task

SYNOPSIS

kern_return_t **mach_port_insert_right**

(mach_port_t	*task,*
mach_port_t	*name,*
mach_port_t	*right,*
mach_msg_type_name_t	*right_type)*;

DESCRIPTION

The **mach_port_insert_right** function inserts into *task* the caller's right for a port, using a specified name for the right in the target task.

The specified *name* can't be one of the reserved values MACH_PORT_NULL or MACH_PORT_DEAD. The *right* can't be MACH_PORT_NULL or MACH_PORT_DEAD.

The argument *right_type* specifies a right to be inserted and how that right should be extracted from the caller. It should be a value appropriate for *msgt_-name*; see **mach_msg**.

If *right_type* 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 holds send or receive rights for the port, then *name* should denote those rights in the target. Otherwise, *name* should be unused in the target. 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 *right_type* is MACH_MSG_TYPE_MAKE_SEND_ONCE or MACH_MSG_-TYPE_MOVE_SEND_ONCE, then a send-once right is inserted. The *name* should be unused in the target. The target gains a send-once right.

If *right_type* is MACH_MSG_TYPE_MOVE_RECEIVE, then a receive right is inserted. If the target already holds send rights for the port, then *name* should denote those rights in the target. Otherwise, *name* should be unused in the target. The receive right is moved into the target task.

PARAMETERS

task

[in scalar] The task which gets the caller's right.

name
: [in scalar] The name by which *task* will know the right.

right
: [in scalar] The port right.

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

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_VALUE
: *name* was MACH_PORT_NULL or MACH_PORT_DEAD.

KERN_NAME_EXISTS
: *name* already denoted a right.

KERN_INVALID_VALUE
: *right* was not a port right.

KERN_INVALID_CAPABILITY
: *right* was null or dead.

KERN_UREFS_OVERFLOW
: Inserting the right would overflow *name*'s user-reference count.

KERN_RIGHT_EXISTS
: *task* already had rights for the port, with a different name.

KERN_RESOURCE_SHORTAGE
: The kernel ran out of memory.

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	*task,*
mach_port_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, and send-once rights may only have one user reference. The resulting number of user references can't be negative. If the resulting number of user references is zero, the effect is to de-allocate the right. For dead names and send rights, 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**, with the exception that **mach_port_destroy** simultaneously destroys all the rights denoted by a name, while **mach_port_mod_refs** can only destroy one right. The name will be available for reuse if it only denoted the one right.

PARAMETERS

task
: [in scalar] The task holding the right.

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

right
: [in scalar] The type of right / entity 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.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_VALUE
right was invalid.

KERN_INVALID_NAME
name did not denote a right.

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

KERN_INVALID_VALUE
The user-reference count would become negative.

KERN_UREFS_OVERFLOW
The user-reference count would overflow.

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 *task,*
mach_port_t *member,*
mach_port_t *after)*;

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

task
[in scalar] The task holding the port set and receive right.

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

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

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
member did not denote a right.

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

KERN_INVALID_NAME
after did not denote a right.

KERN_INVALID_RIGHT
after denoted a right, but not a port set.

KERN_NOT_IN_SET
after was MACH_PORT_NULL, but *member* wasn't currently in a port set.

RELATED INFORMATION

Functions: **mach_port_get_set_status**, **mach_port_get_receive_status**.

mach_port_names

Function — Return information about a task's port name space

SYNOPSIS

```
kern_return_t mach_port_names
                (mach_port_t                    task,
                 mach_port_array_t*             names,
                 mach_msg_type_number_t*        ncount,
                 mach_port_type_array_t*        types,
                 mach_msg_type_number_t*        tcount);
```

DESCRIPTION

The **mach_port_names** returns information about *task*'s port name space. It returns *task*'s currently active names, which represent some port, port set, or dead name right. For each name, it also returns what type of rights *task* holds (the same information returned by **mach_port_type**).

PARAMETERS

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

names
: [out pointer to dynamic array of *mach_port_t*] The names of the ports, port sets, and dead names in the task's port name space, in no particular order.

ncount
: [out scalar] The number of names returned.

types
: [out pointer to dynamic array of *mach_port_type_t*] The type of each corresponding name. Indicates what kind of rights the task holds with that name.

tcount
: [out scalar] Should be the same as *ncount*.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_type**, **vm_deallocate**.

mach_port_rename

Function — Change a task's name for a right

SYNOPSIS

kern_return_t **mach_port_rename**
(mach_port_t *task,*
mach_port_t *old_name,*
mach_port_t *new_name)*;

DESCRIPTION

The **mach_port_rename** function changes the name by which a port, port set, or dead name is known to *task*. *new_name* must not already be in use, and it can't be the distinguished values MACH_PORT_NULL and MACH_-PORT_DEAD.

PARAMETERS

task
[in scalar] The task holding the port right.

old_name
[in scalar] The original name of the port right.

new_name
[in scalar] The new name for the port right.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
old_name did not denote a right.

KERN_INVALID_VALUE
new_name was MACH_PORT_NULL or MACH_PORT_DEAD.

KERN_NAME_EXISTS
new_name already denoted a right.

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_names**.

mach_port_request_notification

Function — Request a notification of a port event

SYNOPSIS

kern_return_t **mach_port_request_notification**

(mach_port_t	*task,*
mach_port_t	*name,*
mach_msg_id_t	*variant,*
mach_port_mscount_t	*sync,*
mach_port_t	*notify,*
mach_msg_type_name_t	*notify_type,*
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.

The *variant* argument takes the following values:

MACH_NOTIFY_PORT_DESTROYED
: *sync* must be zero. The *name* must specify a receive right, and the call requests a port-destroyed notification for the receive right. If the receive right were to have been destroyed, say by **mach_port_destroy**, then instead the receive right will be sent in a port-destroyed notification to the registered send-once right.

(Note: This feature is currently planned for deletion.)

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 which carries the name of the right will be sent to the registered send-once right. If *sync* is non-zero, the *name* may specify a dead name, and a dead-name notification is immediately generated.

Whenever a dead-name notification is generated, the user reference count of the dead name is incremented. For example, a send right with two user refs has a registered dead-name request. If the port is destroyed, the send right turns into a dead name with three user refs (instead of two), and a dead-name notification is generated.

If the name is made available for reuse, perhaps because of **mach_-port_destroy** or **mach_port_mod_refs**, or the name denotes a send-once right which has a message sent to it, then the registered send-once right is used to generate a port-deleted notification instead.

MACH_NOTIFY_NO_SENDERS

The call requests a no-senders notification. *name* must specify a receive right. If the receive right's make-send count is greater than or equal to the sync value, and it has no extant send rights, than an immediate no-senders notification is generated. Otherwise the notification is generated when the receive right next loses its last extant send right. In either case, any previously registered send-once right is returned.

The no-senders notification carries the value the port's make-send count had when it was generated. The make-send count is incremented whenever MACH_MSG_TYPE_MAKE_SEND is used to create a new send right from the receive right. The make-send count is reset to zero when the receive right is carried in a message.

(Note: Currently, moving a receive right does not affect any extant no-senders notifications. It is currently planned to change this so that no-senders notifications are canceled, with a send-once notification sent to indicate the cancelation.)

PARAMETERS

task

[in scalar] The task holding the specified right.

name

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

variant

[in scalar] The type of notification.

sync

[in scalar] Some variants use this value to overcome race conditions.

notify

[in scalar] A send-once right, to which the notification will be sent.

notify_type

[in scalar] IPC type of the sent right; either MACH_MSG_TYPE_-MAKE_SEND_ONCE or MACH_MSG_TYPE_MOVE_SEN-D_ONCE.

previous

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

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_VALUE
variant was invalid.

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted an invalid right.

KERN_INVALID_CAPABILITY
notify was invalid.

When using MACH_NOTIFY_PORT_DESTROYED:

KERN_INVALID_VALUE
sync was not zero.

When using MACH_NOTIFY_DEAD_NAME:

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

KERN_INVALID_ARGUMENT
name denotes a dead name, but *sync* is zero or *notify* is null.

KERN_UREFS_OVERFLOW
name denotes a dead name, but generating an immediate dead-name notification would overflow the name's user-reference count.

RELATED INFORMATION

Functions: **mach_port_get_receive_status**.

mach_port_set_mscount

Function — Changes the make-send count of a port

SYNOPSIS

```
kern_return_t mach_port_set_mscount
                (mach_port_t                 task,
                 mach_port_t                 name,
                 mach_port_mscount_t         mscount);
```

DESCRIPTION

The **mach_port_set_mscount** function changes the make-send count of *task*'s receive right named *name*. All values for *mscount* are valid.

PARAMETERS

task
: [in scalar] The task owning the receive right.

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

mscount
: [in scalar] New value for the make-send count for the receive right.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

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_get_receive_status**, **mach_port_set_qlimit**.

mach_port_set_qlimit

Function — Changes the queue limit of a port

SYNOPSIS

kern_return_t **mach_port_set_qlimit**

(mach_port_t	*task,*
mach_port_t	*name,*
mach_port_msgcount_t	*qlimit)*;

DESCRIPTION

The **mach_port_set_qlimit** function changes the queue limit of *task*'s receive right named *name*. Valid values for *qlimit* are between zero and MACH_-PORT_QLIMIT_MAX (defined in **mach.h**), inclusive.

PARAMETERS

task
: [in scalar] The task owning the receive right.

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

qlimit
: [in scalar] The number of messages which may be queued to this port without causing the sender to block.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
: *name* did not denote a right.

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

KERN_INVALID_VALUE
: *qlimit* was invalid.

RELATED INFORMATION

Functions: **mach_port_get_receive_status**, **mach_port_set_mscount**.

mach_port_set_seqno

Function — Changes the sequence number of a port

SYNOPSIS

kern_return_t **mach_port_set_seqno**

(mach_port_t	*task,*
mach_port_t	*name,*
mach_port_seqno_t	*seqno)*;

DESCRIPTION

The **mach_port_set_seqno** function changes the sequence number of *task*'s receive right named *name*.

PARAMETERS

task
: [in scalar] The task owning the receive right.

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

seqno
: [in scalar] The sequence number that the next message received from the port will have.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

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_get_receive_status**

mach_port_type

Function — Return information about a task's port name

SYNOPSIS

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

DESCRIPTION

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

MACH_PORT_TYPE_SEND
: The name denotes a send right.

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_PORT_SET
: The name denotes a port set.

MACH_PORT_TYPE_DEAD_NAME
: The name is a dead name.

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

MACH_PORT_TYPE_MAREQUEST
: A msg-accepted request for the right is pending. (Note: This feature is planned for deletion.)

MACH_PORT_TYPE_COMPAT
: The port right was created in the compatibility mode.

PARAMETERS

task
: [in scalar] 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, port set, or dead name.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_names**, **mach_port_get_receive_status**, **mach_port_get_set_status**.

mach_ports_lookup

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

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

target_task
: [in scalar] The task whose currently registered ports are to be returned.

init_port_set
: [out pointer to dynamic array of *mach_port_t*] The returned array of ports.

init_port_count
: [out scalar] The number of ports in the array.

RETURN VALUE

KERN_SUCCESS
: The array of registered ports has been returned.

RELATED INFORMATION

Functions: **mach_ports_register**.

mach_ports_register

Function — Registers an array of well-known system ports

SYNOPSIS

```
kern_return_t mach_ports_register
        (mach_port_t                 target_task,
         mach_port_array_t           init_port_set,
         mach_msg_type_number_t      init_port_array_count);
```

DESCRIPTION

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

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

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

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

PARAMETERS

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

init_port_set
: [in pointer to array of *mach_port_t*] The array of ports to register.

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

NOTES

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

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

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

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

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

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

RETURN VALUE

KERN_SUCCESS
: The ports have been registered for the task.

KERN_INVALID_ARGUMENT
: The number of ports exceeds the allowed maximum.

RELATED INFORMATION

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

mach_reply_port

System Trap— Creates a port for the task

LIBRARY

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

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. 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 the next chapter.

vm_allocate

Function — Allocates a region of virtual memory

SYNOPSIS

```
kern_return_t vm_allocate
        (mach_port_t                target_task,
         vm_address_t*              address,
         vm_size_t                  size,
         boolean_t                  anywhere);
```

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. The physical memory is not allocated until an executing thread references the new virtual memory.

If *anywhere* is true, the returned *address* will be at a page boundary and *size* will be rounded up to an integral number of pages. Otherwise, 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 allocated may be greater than *size*. Use **vm_statistics** to find the current virtual page size.

Use the **mach_task_self** function to return the caller's value for *target_task*. This macro returns the task kernel port for the caller.

Initially, there are no access restrictions on any of the pages of the newly allocated region. Child tasks inherit the new region as a copy.

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

PARAMETERS

target_task
: [in scalar] The task in whose address space the region is to be allocated.

address
: [pointer to in/out scalar] The starting address for the region. If there is not enough room following the address, the kernel does not allocate the region. The kernel returns the starting address actually used for the allocated region.

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

anywhere

[in scalar] Placement indicator. If false, the kernel allocates the region starting at *address*. If true, the kernel allocates the region wherever enough space is available within the address space. The kernel returns the starting address actually used in *address*.

NOTES

For languages other than C, use the **vm_statistics** and **mach_task_self** functions to return the task's kernel port (for *target_task*).

A region is a continuous range of addresses bounded by a start address and an end address. Regions consist of pages that have different protection or inheritance characteristics.

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.

RETURN VALUE

KERN_SUCCESS

The new region has been allocated.

KERN_INVALID_ADDRESS

The specified address is illegal.

KERN_NO_SPACE

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

RELATED INFORMATION

Functions: **task_get_special_port**, **vm_deallocate**, **vm_inherit**, **vm_protect**, **vm_region**, **vm_statistics**.

vm_copy

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

SYNOPSIS

```
kern_return_t vm_copy
        (mach_port_t         target_task,
         vm_address_t        source_address,
         vm_size_t           count,
         vm_address_t        dest_address);
```

DESCRIPTION

The **vm_copy** function copies a source region to a destination region within a task's virtual memory. It is 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

target_task
: [in scalar] The task whose memory is to be copied.

source_address
: [in scalar] The starting address for the source region. The address must be on a page boundary.

count
: [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 must be on a page boundary.

RETURN VALUE

KERN_SUCCESS
: The memory region has been copied.

KERN_INVALID_ARGUMENT
: Either an address does not start on a page boundary or the count does not convert to an integral number of pages.

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

KERN_INVALID_ADDRESS
: An address is illegal or specifies a non-allocated region, or there is not enough memory following one of the addresses.

RELATED INFORMATION

Functions: **vm_protect**, **vm_read**, **vm_write**, **vm_statistics**.

vm_deallocate

Function — De-allocates a region of virtual memory

SYNOPSIS

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

DESCRIPTION

The **vm_deallocate** function de-allocates a region of virtual memory in the specified task's address space.

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 de-allocated may be greater than *size*. Use **vm_statistics** to find the current virtual page size.

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

vm_deallocate affects only *target_task*. Other tasks that have access to the de-allocated memory can continue to reference it.

PARAMETERS

target_task
: [in scalar] The task in whose address space the region is to be de-allocated.

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

size
: [in scalar] The number of bytes to de-allocate.

RETURN VALUE

KERN_SUCCESS
: The region has been de-allocated.

KERN_INVALID_ADDRESS
: The address is illegal or specifies a non-allocated region.

RELATED INFORMATION

Functions: **mach_msg**, **vm_allocate**, **vm_statistics**.

vm_inherit

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

SYNOPSIS

```
kern_return_t vm_inherit
        (mach_port_t          target_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

Because inheritance applies to virtual pages, 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 **vm_statistics** 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. This is the only way that two tasks can share memory (other than through the use of an external memory manager; see **vm_map**).

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

PARAMETERS

target_task
: [in scalar] 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.

RETURN VALUE

KERN_SUCCESS
The new inheritance has been set for the region.

KERN_INVALID_ADDRESS
The address is illegal or specifies a non-allocated region.

RELATED INFORMATION

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

vm_machine_attribute

Function — Sets and gets special attributes of a memory region

SYNOPSIS

```
kern_return_t vm_machine_attribute
        (mach_port_t                    target_task,
         vm_address_t                   address,
         vm_size_t                      size,
         vm_machine_attribute_t         attribute,
         vm_machine_attribute_val_t*    value);
```

DESCRIPTION

The **vm_machine_attribute** function gets and sets special attributes of the memory region implemented by the implementations underlying **pmap** module. These attributes are properties such as cachability, migrability and replicability. The behavior of this function is machine dependent.

PARAMETERS

target_task
[in scalar] The task in whose address space the memory object is to be manipulated.

address
[in scalar] The starting address for the memory region. The granularity of rounding of this value to page boundaries is implementation dependent.

size
[in scalar] The number of bytes in the region. The granularity of rounding of this value to page boundaries is implementation dependent.

attribute
[in scalar] The name of the attribute to be get/set. Possible values are:

MATTR_CACHE
Cachability

MATTR_MIGRATE
Migratability

MATTR_REPLICATE
Replicability

value

[pointer to in/out scalar] The new value for the attribute. The old value is also returned in this variable.

MATTR_VAL_OFF
(generic) turn attribute off

MATTR_VAL_ON
(generic) turn attribute on

MATTR_VAL_GET
(generic) return current value

MATTR_VAL_CACHE_FLUSH
flush from all caches

MATTR_VAL_DCACHE_FLUSH
flush from data caches

MATTR_VAL_ICACHE_FLUSH
flush from instruction caches

RETURN VALUE

KERN_SUCCESS
The memory object has been modified.

KERN_INVALID_ARGUMENT
An illegal argument was specified.

vm_map

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

SYNOPSIS

```
kern_return_t vm_map
        (mach_port_t                    target_task,
         vm_address_t*                  address,
         vm_size_t                      size,
         vm_address_t                   mask,
         boolean_t                      anywhere,
         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 *target_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 *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 in 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 use the same page alignment for *offset* and 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.

PARAMETERS

target_task
: [in scalar] The task to whose address space the memory object is to be mapped.

address
: [pointer to in/out scalar] The starting address for the mapped object. If the address is not at the beginning of a virtual page, the kernel rounds it up to the next page boundary. 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 cannot be turned on in the starting address.

anywhere
: [in scalar] Placement indicator. If false, the kernel allocates the object's region starting at *address*. If true, the kernel allocates the region anywhere at or following *address* that there is enough space available within the address space. The kernel returns the starting address actually used in *address*.

memory_object
: [in scalar] The port naming the abstract memory object. 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.

copy
: [in scalar] Copy indicator. If true, the kernel copies the region for the memory object to the specified task's address space. If false, the region is mapped read-write.

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 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.

The kernel rounds the starting address up to the next page boundary. Note that this is different from **vm_allocate**, in which the starting address is rounded down to the previous page boundary.

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_SUCCESS
The memory object has been mapped.

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

KERN_INVALID_ARGUMENT
An illegal argument was specified.

RELATED INFORMATION

Functions: **memory_object_init**, et al., **vm_allocate**.

vm_protect

Function — Sets access privileges for a region of virtual memory

SYNOPSIS

```
kern_return_t vm_protect
        (mach_port_t            target_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 protected may be greater than *size*. Use **vm_statistics** 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.

PARAMETERS

target_task
: [in scalar] 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 scalar] 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 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

KERN_SUCCESS
The new protection has been set for the region.

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

KERN_INVALID_ADDRESS
The address is illegal or specifies a non-allocated region.

RELATED INFORMATION

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

vm_read

Function — Reads a task's virtual memory

SYNOPSIS

```
kern_return_t vm_read
        (mach_port_t                  target_task,
         vm_address_t                 address,
         vm_size_t                    size,
         vm_offset_t*                 data,
         mach_msg_type_number_t*      data_count);
```

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

target_task
: [in scalar] The task whose memory is to be read.

address
: [in scalar] The address at which to start the read. This address must name a page boundary.

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.

data_count
: [out scalar] The number of bytes in the returned array. The count converts to an integral number of pages.

RETURN VALUE

KERN_SUCCESS
: The memory has been read.

KERN_INVALID_ARGUMENT
: Either the address does not start on a page boundary or the size does not convert to an integral number of pages.

KERN_NO_SPACE
: There is not enough room in the calling task's address space to allocate the region for the returned data.

KERN_PROTECTION_FAILURE
: The specified region in the target task is protected against reading.

KERN_INVALID_ADDRESS
: The address is illegal or specifies a non-allocated region, or there are less than *size* bytes of data following the address.

RELATED INFORMATION

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

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_prot_t*           protection,
         vm_prot_t*           max_protection,
         vm_inherit_t*        inheritance,
         boolean_t*           shared,
         mach_port_t*         object_name,
         vm_offset_t*         offset);
```

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 scalar] 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.

protection
: [out scalar] The current protection for the region.

max_protection
: [out scalar] The maximum protection allowed for the region.

inheritance
: [out scalar] The inheritance attribute for the region.

shared
: [out scalar] Shared indicator. If true, the region is shared by another task. If false, the region is not shared.

object_name
: [out scalar] The name of a send right to the name port for the memory object associated with the region. See **memory_object_init**.

offset
: [out scalar] The region's offset into the memory object. The region begins at this offset.

RETURN VALUE

KERN_SUCCESS
: A region has been located and its information returned.

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

RELATED INFORMATION

Functions: **vm_allocate**, **vm_deallocate**, **vm_inherit**, **vm_protect, memory_object_init**, et al.

vm_statistics

Function — Returns statistics on the kernel's use of virtual memory

SYNOPSIS

```
kern_return_t vm_statistics
        (mach_port_t                    target_task,
         vm_statistics_data_t*          vm_stats);
```

DESCRIPTION

The **vm_statistics** function returns statistics on the kernel's use of virtual memory from the time the kernel was booted.

See **vm_statistics** for a description of the structure used.

For related information for a specific task, use **task_info**.

PARAMETERS

target_task
: [in scalar] The task that is requesting the statistics.

vm_stats
: [out structure] The structure in which the statistics will be returned.

RETURN VALUE

KERN_SUCCESS
: The statistics have been returned.

KERN_INVALID_HOST
: The host is null.

KERN_RESOURCE_SHORTAGE
: The kernel could not allocate sufficient memory.

RELATED INFORMATION

Functions: **task_info**.

Data Structures: **vm_statistics**.

vm_wire

Function — Specifies the pageability of a region of virtual memory

LIBRARY

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

SYNOPSIS

```
kern_return_t vm_wire
        (mach_port_t                 host_priv,
         mach_port_t                 target_task,
         vm_address_t                address,
         vm_size_t                   size,
         vm_prot_t                   wired_access);
```

DESCRIPTION

The **vm_wire** function sets the pageability privileges for a region within the specified task's address space. *wired_access* specifies an access attribute which is interpreted to specify whether the region can be paged.

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 **vm_statistics** to find the current virtual page size.

This call is directed to the privileged host port on which *target_task* executes because of the privileged nature of committing physical memory.

PARAMETERS

host_priv
: [in scalar] The host control port for the host on which *target_task* executes.

target_task
: [in scalar] 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.

wired_access
: [in scalar] The pageability of the region. Valid values are:

VM_PROT_NONE
: Un-wire (allow to be paged) the region of memory.

Any other value specifies that the region is to be wired and that the target task must have at least the specified amount of access to the region.

RETURN VALUE

KERN_SUCCESS
: The new pageability has been set for the region.

KERN_INVALID_HOST
: The privileged host port was not specified.

KERN_INVALID_ADDRESS
: The address is illegal or specifies a non-allocated region.

KERN_INVALID_VALUE
: An invalid value for *wired_access* was specified.

RELATED INFORMATION

Functions: **thread_wire**.

vm_write

Function — Writes data to a task's virtual memory

SYNOPSIS

kern_return_t **vm_write**	
(mach_port_t	*target_task,*
vm_address_t	*address,*
vm_offset_t	*data,*
mach_msg_type_number_t	*data_count*);

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.

Use **vm_statistics** to find the current virtual page size.

PARAMETERS

target_task
: [in scalar] The task whose memory is to be written.

address
: [in scalar] The address at which to start the write. The starting address must be on a page boundary.

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

data_count
: [in scalar] The number of bytes in the array. The size of the array must convert to an integral number of pages.

RETURN VALUE

KERN_SUCCESS
: The memory has been written.

KERN_INVALID_ARGUMENT
: Either the address does not start on a page boundary or *data_count* does not convert to an integral number of pages.

KERN_PROTECTION_FAILURE
: The specified region in the target task is protected against writing.

KERN_INVALID_ADDRESS
: The address is illegal or specifies a non-allocated region, or there are less than *data_count* bytes available following the address.

RELATED INFORMATION

Functions: **vm_copy**, **vm_protect**, **vm_read**, **vm_statistics**.

CHAPTER 5 External Memory Management Interface

This chapter discusses the specifics of the kernel's external memory management interfaces. Interfaces that relate to the basic use of virtual memory for a task appear in the previous chapter.

default_pager_info

Function —Return default partition information

LIBRARY

libmach.a only

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

SYNOPSIS

kern_return_t **default_pager_info**
(mach_port_t *pager,*
vm_size_t* *total,*
vm_size_t* *free)*;

DESCRIPTION

The **default_pager_info** function returns information concerning the default pager's default paging partition.

The default memory manager port can be obtained by calling **vm_set_default_-memory_manager** with the host control port, specifying the "new" pager port as MACH_PORT_NULL.

PARAMETERS

pager
[in scalar] A port to the default memory manager.

total
[out scalar] Total size of the default partition.

free
[out scalar] Free space in the default partition.

RETURN VALUE

KERN_SUCCESS
Information returned.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**.

default_pager_object_create

Function — Create a memory object managed by the default pager

LIBRARY

libmach.a only

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

SYNOPSIS

```
kern_return_t default_pager_object_create
                (mach_port_t                    pager,
                 memory_object_t*               memory_object,
                 vm_size_t                      object_size);
```

DESCRIPTION

The **default_pager_object_create** function returns an object, backed by the default pager, which is suitable for use with **vm_map**. This memory object has the same properties as does a memory object provided by **vm_allocate**: its initial contents are zero and the backing contents are temporary in that they do not persist after the memory object is destroyed. The memory object is suitable for use as non-permanent shared memory.

The default memory manager port can be obtained by calling **vm_set_default_memory_manager** with the host control port, specifying the "new" pager port as MACH_PORT_NULL.

PARAMETERS

pager
: [in scalar] A port to the default memory manager.

memory_object
: [out scalar] The abstract memory object port for the memory object.

object_size
: [in scalar] The maximum size for the memory object.

RETURN VALUE

KERN_SUCCESS
: Memory object created.

RELATED INFORMATION

Functions: **vm_map**, **vm_set_default_memory_manager**.

memory_object_change_attributes

Function — Changes various performance related attributes

SYNOPSIS

```
kern_return_t memory_object_change_attributes
                (mach_port_t                        memory_control,
                 boolean_t                          may_cache_object,
                 memory_object_copy_strategy_t      copy_strategy,
                 mach_port_t                        reply_to);
```

DESCRIPTION

The **memory_object_change_attributes** function sets various performance-related attributes for the specified memory object, so as to:

- Retain data from a memory object even after all address space mappings have been de-allocated (*may_cache_object* parameter).
- Perform optimizations for virtual memory copy operations (*copy_strategy* parameter).

PARAMETERS

memory_control

[in scalar] 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_init** call.

may_cache_object

[in scalar] Cache indicator. If true, the kernel can cache data associated with the memory object, even if virtual memory references to it are removed.

copy_strategy

[in scalar] How the kernel should handle copying of regions associated with the memory object. 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_CALL

Notify the memory manager (via **memory_object_copy**) before copying any 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.

MEMORY_OBJECT_COPY_TEMPORARY
Mark the object as temporary. This had the same effect as the MEMORY_OBJECT_COPY_DELAY strategy and has the additional attribute that when the last mapping of the memory object is removed, the object is destroyed without returning any in-memory pages.

reply_port
[in scalar] A port to which a reply (**memory_object_change_completed**) is to be sent indicating the completion of the attribute change. Such a reply would be useful if the cache attribute is turned off, since such a change, if the memory object is no longer mapped, may result in the object being terminated, or if the copy strategy is changed, which may result in additional page requests.

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.

RETURN VALUE

KERN_SUCCESS
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_completed**, **memory_object_copy**, **memory_object_get_attributes**, **memory_object_ready**, **memory_object_set_attributes** (old form).

memory_object_change_completed

Server Interface — Indicates completion of an attribute change call

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_change_completed**

(mach_port_t	*memory_object,*
boolean_t	*may_cache_object,*
memory_object_copy_strategy_t	*copy_strategy*);

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. A response is generated so that the manager can synchronize with changes to the copy strategy (which affects the manner in which pages will be requested) and a termination message possibly resulting from un-cacheing a not-mapped object.

SEQUENCE NUMBER FORM

seqnos_memory_object_change_completed

kern_return_t **seqnos_memory_object_change_completed**

(mach_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
boolean_t	*may_cache_object,*
memory_object_copy_strategy_t	*copy_strategy*);

PARAMETERS

memory_object

[in scalar] The abstract memory object 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 port named in the **memory_object_change_attributes** call.

may_cache_object
[in scalar] The new cache attribute.

copy_strategy
[in scalar] The new copy strategy.

NOTES

No memory cache control port is supplied in this call because the attribute change may cause termination of the object leading to what would be an invalid cache port.

RETURN VALUE

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_change_attributes**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_copy

Server Interface — Indicates that a memory object has been copied

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_copy
                (mach_port_t                   old_memory_object,
                 memory_object_control_t       old_memory_control,
                 vm_offset_t                   offset,
                 vm_size_t                     length,
                 mach_port_t                   new_memory_object);
```

DESCRIPTION

A **memory_object_copy** function is called as the result of a message from the kernel indicating that the kernel has copied the specified region within the old memory object.

This call includes only the new abstract memory object port itself. The kernel will subsequently issue a **memory_object_init** call on the new abstract memory object after it has prepared the currently cached pages of the old object. When the memory manager receives the **memory_object_init** call, it is expected to reply with the **memory_object_ready** call. The kernel uses the new abstract memory object, memory cache control, and memory cache name ports to refer to the new copy.

The kernel makes the **memory_object_copy** call only if:

- The memory manager had previously set the old object's copy strategy attribute to MEMORY_OBJECT_COPY_CALL (using **memory_object_change_attributes** or **memory_object_ready**).
- A user of the old object has asked the kernel to copy it.

Cached pages from the old memory object at the time of the copy are handled as follows:

- Readable pages may be copied to the new object without notification and with all access permissions.
- Pages not copied are locked to prevent write access.

The memory manager should treat the new memory object as temporary. In other words, the memory manager should not change the new object's contents or allow it to be mapped in another client. The memory manager can use the **mem-**

ory_object_data_unavailable call to indicate that the appropriate pages of the old object can be used to fulfill a data request.

SEQUENCE NUMBER FORM

seqnos_memory_object_copy

```
kern_return_t seqnos_memory_object_copy
        (mach_port_t                  old_memory_object,
         mach_port_seqno_t            seqno,
         memory_object_control_t      old_memory_control,
         vm_offset_t                  offset,
         vm_size_t                    length,
         mach_port_t                  new_memory_object);
```

PARAMETERS

old_memory_object
: [in scalar] The port that represents the old (copied from) abstract memory object.

seqno
: [in scalar] The sequence number of this message relative to the abstract memory object port.

old_memory_control
: [in scalar] The kernel memory cache control port for the old memory object.

offset
: [in scalar] The offset within the old memory object.

length
: [in scalar] The number of bytes copied, starting at *offset*. The number converts to an integral number of virtual pages.

new_memory_object
: [in scalar] The new abstract memory object created by the kernel. The kernel provides all port rights (including the receive right) for the new memory object.

NOTES

It is possible for a memory manager to receive a **memory_object_data_return** message for a page of the new memory object before receiving any other requests for that data.

RETURN VALUE

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_change_attributes**, **memory_object_data_unavailable**, **memory_object_init**, **memory_object_ready**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_create

Server Interface — Requests transfer of responsibility for a kernel-created memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_create
                (mach_port_t          old_memory_object,
                 mach_port_t          new_memory_object,
                 vm_size_t            new_object_size,
                 mach_port_t          new_control,
                 mach_port_t          new_name,
                 vm_size_t            new_page_size);
```

DESCRIPTION

A **memory_object_create** function is called as the result of a message from the kernel requesting that the default memory manager accept responsibility for the new memory object created by the kernel. The kernel makes this call only to the system default memory manager.

The new memory object initially consists of zero-filled pages. Only memory pages that are actually written are provided to the memory manager. When processing **memory_object_data_request** calls from the kernel, the default memory manager must use **memory_object_data_unavailable** for any pages that have not been written previously.

The kernel does not expect a reply to this call. The kernel assumes that the default memory manager will be ready to handle data requests to this object and does not need the confirmation of a **memory_object_ready** call.

SEQUENCE NUMBER FORM

seqnos_memory_object_create

```
kern_return_t seqnos_memory_object_create
                (mach_port_t          old_memory_object,
                 mach_port_seqno_t    seqno,
                 mach_port_t          new_memory_object,
                 vm_size_t            new_object_size,
                 mach_port_t          new_control,
                 mach_port_t          new_name,
                 vm_size_t            new_page_size);
```

PARAMETERS

old_memory_object
: [in scalar] An existing abstract memory object provided by the default memory manager.

seqno
: [in scalar] The sequence number of this message relative to the old abstract memory object port.

new_memory_object
: [in scalar] The port representing the new abstract memory object created by the kernel. The kernel provides all port rights (including the receive right) for the new memory object.

new_object_size
: [in scalar] The maximum size for the new object, in bytes.

new_control
: [in scalar] The memory cache port to be used by the memory manager when making cache management requests for the new object.

new_name
: [in scalar] The memory cache name port used by the kernel to refer to the new memory object data in response to **vm_region** calls.

new_page_size
: [in scalar] The page size used by the kernel. All calls involving this kernel must use data sizes that are integral multiples of this page size.

NOTES

The kernel requires memory objects to provide temporary backing storage for zero-filled memory created by **vm_allocate** calls, issued by both user tasks and the kernel itself. The kernel allocates an abstract memory object port to represent the temporary backing storage and uses **memory_object_create** to pass the new memory object to the default memory manager, which provides the storage.

The default memory manager is a trusted system component that is identified to the kernel at system initialization time. The default memory manager can also be changed at run time using the **vm_set_default_memory_manager** call.

The contents of a kernel-created (as opposed to a user-created) memory object can be modified only in main memory. The default memory manager must not change the contents of a temporary memory object, or allow unrelated tasks to access the memory object, control, or name port.

The kernel can provide the maximum size of a temporary memory object because the object cannot be mapped by another user task.

RETURN VALUE

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_data_initialize**, **memory_object_data_unavailable**, **memory_object_default_server**, **seqnos_memory_object_default_server**.

memory_object_data_error

Function — Indicates no data for a memory object

SYNOPSIS

```
kern_return_t memory_object_data_error
                (mach_port_t          memory_control,
                 vm_offset_t          offset,
                 vm_size_t            size,
                 kern_return_t        reason);
```

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

memory_control
: [in scalar] 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_init** call.

offset
: [in scalar] The offset within the memory object, in bytes.

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

reason
: [in scalar] Reason for the error. The value could be a POSIX error code for a hardware error.

NOTES

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

RETURN VALUE

KERN_SUCCESS
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_initialize

Server Interface — Writes initial data back to a temporary memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_data_initialize
        (mach_port_t          memory_object,
         mach_port_t          memory_control,
         vm_offset_t          offset,
         vm_offset_t          data,
         vm_size_t            data_count);
```

DESCRIPTION

A **memory_object_data_initialize** function is called as the result of a kernel message providing the default memory manager with initial data for a kernel-created memory object. If the memory manager already has supplied data (by a previous **memory_object_data_initialize** or **memory_object_data_return**), it should ignore this call. Otherwise, the call behaves the same as the **memory_object_data_return** call.

The kernel makes this call only to the default memory manager and only on temporary memory objects that it has created with **memory_object_create**. Note that the kernel does not make this call on objects created via **memory_object_copy**.

SEQUENCE NUMBER FORM

seqnos_memory_object_data_initialize

```
kern_return_t seqnos_memory_object_data_initialize
        (mach_port_t          memory_object,
         mach_port_seqno_t    seqno,
         mach_port_t          memory_control,
         vm_offset_t          offset,
         vm_offset_t          data,
         vm_size_t            data_count);
```

PARAMETERS

memory_object
: [in scalar] The abstract memory object port that represents the memory object data, as supplied by the kernel in a **memory_object_create** call.

seqno
[in scalar] The sequence number of this message relative to the abstract memory object port.

memory_control
[in scalar] The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset
[in scalar] The offset within the memory object.

data
[in pointer to dynamic array of bytes] The data that has been modified while cached in physical memory.

data_count
[in scalar] The number of bytes to be written, starting at *offset*. The number converts to an integral number of memory object pages.

RETURN VALUE

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_create**, **memory_object_data_return**, **memory_-object_default_server**, **seqnos_memory_object_default_server**.

memory_object_data_provided

Function — Supplies data for a region of a memory object (old form)

SYNOPSIS

```
kern_return_t memory_object_data_provided
                (mach_port_t                    memory_control,
                 vm_offset_t                    offset,
                 vm_offset_t                    data,
                 vm_size_t                      data_count,
                 vm_prot_t                      lock_value);
```

DESCRIPTION

The **memory_object_data_provided** function supplies the kernel with a range of data for the specified memory object. A memory manager normally provides data only in response to a **memory_object_data_request** call from the kernel.

PARAMETERS

memory_control
: [in scalar] 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_init** call.

offset
: [in scalar] The offset within the memory object, in bytes.

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

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

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_READ
: Prohibits read access.

VM_PROT_WRITE
: Prohibits write access.

VM_PROT_EXECUTE
: Prohibits execute access.

VM_PROT_ALL
: Prohibits all forms of access.

NOTES

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

memory_object_data_provided is the old form of **memory_object_data_supply**.

CAUTIONS

A memory manager must be careful when providing 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. Currently, the kernel prohibits the overwriting of live data pages.

RETURN VALUE

KERN_SUCCESS
: 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_data_unavailable**, **memory_object_lock_request**.

memory_object_data_request

Server Interface — Requests data from a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_data_request
        (mach_port_t                memory_object,
         mach_port_t                memory_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_init** 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.

SEQUENCE NUMBER FORM

seqnos_memory_object_data_request

```
kern_return_t seqnos_memory_object_data_request
        (mach_port_t                memory_object,
         mach_port_seqno_t          seqno,
         mach_port_t                memory_control,
         vm_offset_t                offset,
         vm_size_t                  length,
         vm_prot_t                  desired_access);
```

PARAMETERS

memory_object
[in scalar] The abstract memory object 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 abstract memory object port.

memory_control
[in scalar] 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.

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

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_data_error**, **memory_object_data_supply**, **memory_object_data_unavailable**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_data_return

Server Interface — Writes data back to a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_data_return
        (mach_port_t                    memory_object,
         mach_port_t                    memory_control,
         vm_offset_t                    offset,
         vm_offset_t                    data,
         vm_size_t                      data_count,
         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. When the memory manager no longer needs the data (for example, after the data has been written to permanent storage), it should use **vm_deallocate** to release the memory resources.

SEQUENCE NUMBER FORM

seqnos_memory_object_data_return

```
kern_return_t seqnos_memory_object_data_return
        (mach_port_t                    memory_object,
         mach_port_seqno_t              seqno,
         mach_port_t                    memory_control,
         vm_offset_t                    offset,
         vm_offset_t                    data,
         vm_size_t                      data_count,
         boolean_t                      dirty,
         boolean_t                      kernel_copy);
```

PARAMETERS

memory_object
: [in scalar] The abstract memory object 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 abstract memory object port.

memory_control
[in scalar] The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset
[in scalar] The offset within the memory object.

data
[in pointer to dynamic array of bytes] The data that has been evicted from the physical memory cache.

data_count
[in scalar] The number of bytes to be written, starting at *offset*. The number converts to an integral number of memory object pages.

dirty
[in scalar] If TRUE, the pages returned have been modified.

kernel_copy
[in scalar] 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

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_data_supply**, **memory_object_data_write** (old form), **vm_deallocate**, **memory_object_server**, **seqnos_memory_object_server**.

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                memory_control,
         vm_offset_t                offset,
         vm_offset_t                data,
         mach_msg_type_number_t     data_count,
         boolean_t                  deallocate,
         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 normally provides data only in response to a **memory_object_data_request** call from the kernel.

PARAMETERS

memory_control
: [in scalar] 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_init** call.

offset
: [in scalar] The offset within the memory object, in bytes.

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

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

deallocate
: [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_READ
  : Prohibits read access.

  VM_PROT_WRITE
  : Prohibits write access.

  VM_PROT_EXECUTE
  : Prohibits execute access.

  VM_PROT_ALL
  : Prohibits all forms of access.

precious
: [in scalar] 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 scalar] 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 when providing 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. Currently, the kernel prohibits the overwriting of live data pages.

RETURN VALUE

KERN_SUCCESS
: 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_provided** (old form), **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                    memory_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 for this region.

A memory manager can use this call in any of the following situations:

- When the object was created by the kernel (via **memory_object_create**) and the kernel has not yet provided data for the region (via either **memory_object_data_initialize** or **memory_object_data_return**). In this case, the object is a temporary memory object; the memory manager is the default memory manager; and the kernel should provide zero-filled pages for the object.
- When the object was created by a **memory_object_copy**. In this case, the kernel should copy the region from the original memory object.
- When the object is a normal user-created memory object. In this case, the kernel should provide unlocked zero-filled pages for the region.

PARAMETERS

memory_control
: [in scalar] 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_init** or a **memory_object_create** call.

offset
: [in scalar] The offset within the memory object, in bytes.

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

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

RELATED INFORMATION

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

memory_object_data_unlock

Server Interface — Requests access to a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_data_unlock**

(mach_port_t	*memory_object,*
mach_port_t	*memory_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.

SEQUENCE NUMBER FORM

seqnos_memory_object_data_unlock

kern_return_t **seqnos_memory_object_data_unlock**

(mach_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
mach_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_size_t	*length,*
vm_prot_t	*desired_access*);

PARAMETERS

memory_object
: [in scalar] The abstract memory object 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 abstract memory object port.

memory_control
: [in scalar] 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.

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

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_lock_completed**, **memory_object_lock_request**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_data_write

Server Interface — Writes changed data back to a memory object (old form)

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_data_write**

(mach_port_t	*memory_object,*
mach_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_offset_t	*data,*
vm_size_t	*data_count*);

DESCRIPTION

A **memory_object_data_write** function is called as the result of a kernel message providing the memory manager with data that has been modified while cached in physical memory. This old form is used if the memory manager makes the object ready via the old **memory_object_set_attributes** instead of **memory_object_ready**.

The kernel writes back only data that has been modified. When the memory manager no longer needs the data (for example, after the data has been written to permanent storage), it should use **vm_deallocate** to release the memory resources.

SEQUENCE NUMBER FORM

seqnos_memory_object_data_write

kern_return_t **seqnos_memory_object_data_write**

(mach_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
mach_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_offset_t	*data,*
vm_size_t	*data_count*);

PARAMETERS

memory_object

[in scalar] The abstract memory object 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 abstract memory object port.

memory_control

[in scalar] The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset

[in scalar] The offset within the memory object.

data

[in pointer to dynamic array of bytes] The data that has been modified while cached in physical memory.

data_count

[in scalar] The number of bytes to be written, starting at *offset*. The number converts to an integral number of memory object pages.

NOTES

The kernel can flush clean (that is, un-modified) 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

KERN_SUCCESS

This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_data_return**, **memory_object_set_attributes**, **vm_deallocate**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_destroy

Function — Shuts down a memory object

SYNOPSIS

kern_return_t **memory_object_destroy**
(mach_port_t *memory_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 pass to the memory manager all rights to the memory object port, the memory control port, and the memory name port.

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

memory_control
[in scalar] 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_init** 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

KERN_SUCCESS
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_get_attributes

Function — Returns current attributes for a memory object

SYNOPSIS

```
kern_return_t memory_object_get_attributes
                (mach_port_t                          memory_control,
                 boolean_t*                           object_ready,
                 boolean_t*                           may_cache_object,
                 memory_object_copy_strategy_t*       copy_strategy);
```

DESCRIPTION

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

PARAMETERS

memory_control
: [in scalar] 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_init** call.

object_ready
: [out scalar] Ready indicator. If true, the kernel can issue new data and unlock requests on the memory object.

may_cache_object
: [out scalar] Cache indicator. If true, the kernel can cache data associated with the memory object, even if virtual memory references to it are removed.

copy_strategy
: [out scalar] How the kernel should handle copying of regions associated with the memory object. Possible 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_CALL
: Notify the memory manager (via **memory_object_copy**) before copying any 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.

MEMORY_OBJECT_COPY_TEMPORARY
: Mark the object as temporary. This had the same effect as the MEMORY_OBJECT_COPY_DELAY strategy and has the additional attribute that when the last mapping of the memory object is removed, the object is destroyed without returning any in-memory pages.

RETURN VALUE

KERN_SUCCESS
: 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_copy**, **memory_object_ready**.

memory_object_init

Server Interface — Initializes a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_init
                (mach_port_t                    memory_object,
                 mach_port_t                    memory_control,
                 mach_port_t                    memory_object_name,
                 vm_size_t                      memory_object_page_size);
```

DESCRIPTION

A **memory_object_init** 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 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_init** call on the abstract memory object. This call is provided as a convenience to the memory manager, to allow it to initialize data structures and prepare to receive other requests.

In addition to the abstract memory object port itself, the call provides the following two ports:

- A memory cache control port that the memory manager can use to control use of its data by the kernel. The memory manager gets send rights for this port.
- A memory cache name port that the kernel will use to identify the memory object to other tasks.

The kernel holds send rights for the abstract memory object port, and both send and receive rights for the memory cache control and name ports.

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_init** call from each kernel, containing a unique set of control and name ports. Note that each kernel may also use a different page size.

SEQUENCE NUMBER FORM

seqnos_memory_object_init

kern_return_t **seqnos_memory_object_init**

(mach_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
mach_port_t	*memory_control,*
mach_port_t	*memory_object_name,*
vm_size_t	*memory_object_page_size*);

PARAMETERS

memory_object
: [in scalar] The abstract memory object 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 abstract memory object port.

memory_control
: [in scalar] The memory cache control port to be used 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.

memory_object_name
: [in scalar] The memory cache name port used by the kernel to refer to the memory object data in response to **vm_region** calls.

memory_object_page_size
: [in scalar] The page size used by the kernel. All calls involving this kernel must use data sizes that are integral multiples of this page size.

NOTES

When the memory manager is ready to accept data requests for this memory object, it must call **memory_object_ready**. Otherwise, the kernel will not process requests on this object.

RETURN VALUE

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_ready**, **memory_object_terminate**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_lock_completed

Server Interface — Indicates completion of a consistency control call

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_lock_completed
                (mach_port_t                 memory_object,
                 mach_port_t                 memory_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.

SEQUENCE NUMBER FORM

seqnos_memory_object_lock_completed

```
kern_return_t seqnos_memory_object_lock_completed
                (mach_port_t                 memory_object,
                 mach_port_seqno_t           seqno,
                 mach_port_t                 memory_control,
                 vm_offset_t                 offset,
                 vm_size_t                   length);
```

PARAMETERS

memory_object
: [in scalar] The abstract memory object 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 port named in the **memory_object_lock_request** message.

memory_control
: [in scalar] 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.

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

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_lock_request**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_lock_request

Function — Restricts access to memory object data

SYNOPSIS

kern_return_t **memory_object_lock_request**

(mach_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_size_t	*size,*
*m*emory_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

memory_control

[in scalar] 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_init** call.

offset

[in scalar] The offset within the memory object, in bytes.

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. If *should_flush* is TRUE, precious pages will be discarded; otherwise, the kernel 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 scalar] 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_READ

Prohibits read access.

VM_PROT_WRITE

Prohibits write access.

VM_PROT_EXECUTE

Prohibits execute access.

VM_PROT_ALL

Prohibits all forms of access.

reply_to
: [in scalar] 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.

To indicate that an unlock request is invalid (that is, requires permission that can never be granted), the memory manager must first flush the page. When the kernel requests the data again with the higher permission, the memory manager can indicate the error by responding with a call to **memory_object_data_error**.

RETURN VALUE

KERN_SUCCESS
: 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_supply**, **memory_object_data_unlock**, **memory_object_lock_completed**.

memory_object_ready

Function — Marks a memory object is ready to receive paging operations

SYNOPSIS

kern_return_t **memory_object_ready**
(mach_port_t *memory_control,*
boolean_t *may_cache_object,*
memory_object_copy_strategy_t *copy_strategy*);

DESCRIPTION

The **memory_object_ready** function informs the kernel that the manager is ready to receive data or unlock requests on behalf of clients. Performance-related attributes for the specified memory object can also be set at this time. These attributes control whether the kernel is permitted to:

- Retain data from a memory object even after all address space mappings have been de-allocated (*may_cache_object* parameter).
- Perform optimizations for virtual memory copy operations (*copy_strategy* parameter).

PARAMETERS

memory_control
[in scalar] 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_init** call.

may_cache_object
[in scalar] Cache indicator. If true, the kernel can cache data associated with the memory object, even if virtual memory references to it are removed.

copy_strategy
[in scalar] How the kernel should handle copying of regions associated with the memory object. 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_CALL
Notify the memory manager (via **memory_object_copy**) before copying any 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.

MEMORY_OBJECT_COPY_TEMPORARY
: Mark the object as temporary. This had the same effect as the MEMORY_OBJECT_COPY_DELAY strategy and has the additional attribute that when the last mapping of the memory object is removed, the object is destroyed without returning any in-memory pages.

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.

RETURN VALUE

KERN_SUCCESS
: 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_copy**, **memory_object_get_attributes**, **memory_object_init**, **memory_object_set_attributes** (old form).

memory_object_set_attributes

Function — Sets attributes for a memory object (old form)

SYNOPSIS

```
kern_return_t memory_object_set_attributes
        (mach_port_t                        memory_control,
         boolean_t                          object_ready,
         boolean_t                          may_cache_object,
         memory_object_copy_strategy_t      copy_strategy);
```

DESCRIPTION

The **memory_object_set_attributes** function allows the memory manager to set performance-related attributes for the specified memory object. These attributes control whether the kernel is permitted to:

- Make data or unlock requests on behalf of clients (*object_ready* parameter).
- Retain data from a memory object even after all address space mappings have been de-allocated (*may_cache_object* parameter).
- Perform optimizations for virtual memory copy operations (*copy_strategy* parameter).

PARAMETERS

memory_control
: [in scalar] 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_init** call.

object_ready
: [in scalar] Ready indicator. If true, the kernel can issue new data and unlock requests on the memory object.

may_cache_object
: [in scalar] Cache indicator. If true, the kernel can cache data associated with the memory object, even if virtual memory references to it are removed.

copy_strategy
: [in scalar] How the kernel should handle copying of regions associated with the memory object. 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_CALL
: Notify the memory manager (via **memory_object_copy**) before copying any 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.

MEMORY_OBJECT_COPY_TEMPORARY
: Mark the object as temporary. This had the same effect as the MEMORY_OBJECT_COPY_DELAY strategy and has the additional attribute that when the last mapping of the memory object is removed, the object is destroyed without returning any in-memory pages.

NOTES

memory_object_set_attributes is the old form of **memory_object_change_attributes**. When used to change the cache or copy strategy attributes, it has the same effect (with the omission of a possible reply) as **memory_object_change_attributes**. The difference between these two calls is the *ready* attribute. The use of this old call with the *ready* attribute set has the same basic effect as the new **memory_object_ready** call. However, the use of this old call informs the kernel that this is an old form memory manager that expects **memory_object_data_write** messages instead of the new **memory_object_data_return** messages implied by **memory_object_ready**. Changing a memory object to be not ready does not affect data and unlock requests already in progress. Such requests will not be aborted or reissued.

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.

RETURN VALUE

KERN_SUCCESS
: 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_copy**, **memory_object_get_attributes**, **memory_object_init**, **memory_object_ready**.

memory_object_supply_completed

Server Interface — Indicates completion of a data supply call

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_supply_completed**

(mach_port_t	*memory_object,*
mach_port_t	*memory_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.

SEQUENCE NUMBER FORM

seqnos_memory_object_supply_completed

kern_return_t **seqnos_memory_object_supply_completed**

(mach_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
mach_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_size_t	*length*,
kern_return_t	*result*,
vm_offset_t	*error_offset*);

PARAMETERS

memory_object
: [in scalar] The abstract memory object 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 port named in the **memory_object_data_supply** call.

memory_control
: [in scalar] 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

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 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

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_data_supply**, **memory_object_server**, **seqnos_-memory_object_server**.

memory_object_terminate

Server Interface — Relinquishes access to a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_terminate
        (mach_port_t                 memory_object,
         mach_port_t                memory_control,
         mach_port_t           memory_object_name);
```

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 call provides receive rights to the memory cache control and name ports so that the memory manager can destroy the ports (via **mach_port_deallocate**). The kernel also relinquishes its send rights for all three 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.

SEQUENCE NUMBER FORM

seqnos_memory_object_terminate

```
kern_return_t seqnos_memory_object_terminate
        (mach_port_t                 memory_object,
         mach_port_seqno_t                   seqno,
         mach_port_t                memory_control,
         mach_port_t           memory_object_name);
```

PARAMETERS

memory_object
: [in scalar] The abstract memory object 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 abstract memory object port.

memory_control
: [in scalar] 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.

memory_object_name
: [in scalar] The memory cache name port used by the kernel to refer to the memory object data in response to **vm_region** calls.

NOTES

If a client thread calls **vm_map** to map a memory object while the kernel is calling **memory_object_terminate** for the same memory object, the **memory_object_init** call may appear before the **memory_object_terminate** call. This sequence is indistinguishable from the case where another kernel is issuing a **memory_object_init** call. In other words, the control and name ports included in the initialization will be different from those included in the termination. A memory manager must be aware that this sequence can occur even when all mappings of a memory object take place on the same host.

RETURN VALUE

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_destroy**, **memory_object_init**, **mach_port_deallocate**, **memory_object_server**, **seqnos_memory_object_server**.

vm_set_default_memory_manager

Function — Sets the default memory manager.

SYNOPSIS

```
kern_return_t vm_set_default_memory_manager
                (mach_port_t                    host,
                 mach_port_t*                   default_manager);
```

DESCRIPTION

The **vm_set_default_memory_manager** function establishes the default memory manager for a host.

PARAMETERS

host
: [in scalar] The control port naming the host for which the default memory manager is to be set.

default_manager
: [pointer to in/out scalar] A memory manager port to the new default memory manager. If this value is MACH_PORT_NULL, the old memory manager is not changed. The old memory manager port is returned in this variable.

RETURN VALUE

KERN_SUCCESS
: The old default memory port was returned and the new manager established.

KERN_INVALID_ARGUMENT
: The supplied host port is not the host control port.

RELATED INFORMATION

Functions: **memory_object_create**, **vm_allocate**.

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.

catch_exception_raise

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

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t catch_exception_raise
                (mach_port_t          exception_port,
                 mach_port_t          thread,
                 mach_port_t          task,
                 int                  exception,
                 int                  code,
                 int                  subcode);
```

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_special_port** or **task_set_special_port** as the port that responds when the thread takes an exception.

PARAMETERS

exception_port
: [in scalar] The port to which the exception notification was sent.

thread
: [in scalar] The control port to the thread taking the exception.

task
: [in scalar] The control port to 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 *kern_return_t* describing error. *subcode* contains 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* and *subcode* fields.

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

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

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

subcode
: [in scalar] A specific type of *code*.

NOTES

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

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

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

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

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

A thread can have two exception ports active for it: its thread exception port and the task exception port. If an exception message is sent to the thread exception port (if it exists), and a reply message contains a return value other than KERN_SUCCESS, the kernel will then send the exception message to the task exception port. If that exception message receives a reply message with other than a return value of KERN_SUCCESS, the thread is terminated. Note that this behavior cannot be obtained by using the **catch_exception_raise** interface called by **exc_server** and **mach_msg_server**, since those functions will either return a reply message with a KERN_SUCCESS value, or none at all.

RETURN VALUE

KERN_SUCCESS
: The thread is to continue from the point of exception.

Other values indicate that the exception was handled directly and the thread was restarted or terminated by the exception handler.

RELATED INFORMATION

Functions: **exception_raise**, **exc_server**, **thread_abort**, **task_get_special_port**, **thread_get_special_port**, **thread_get_state**, **thread_resume**, **task_set_special_port,** **thread_set_special_port**, **thread_set_state**, **thread_suspend**, **thread_terminate.**

evc_wait

System Trap — Wait for a kernel (device) signalled event

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **evc_wait**
(unsigned int *event*);

DESCRIPTION

The **evc_wait** function causes the invoking thread to wait until the specified kernel (device) generated event occurs. Device drivers (typically mapped devices intended to be supported by user space drivers) may supply an event count service.

The event count service defines one or more event objects, named by task local event IDs. Each of these event objects has an associated event count, initially zero. Whenever the associated event occurs (typically a device interrupt), the event count is incremented. If this count is zero when **evc_wait** is called, the calling thread waits for the next event to occur. Only one thread may be waiting for the event to occur. If the count is non-zero when **evc_wait** is called, the count is simply decremented without causing the thread to wait. The event count guarantees that no events are lost.

PARAMETERS

event
: [in scalar] The task local event ID of the kernel event object.

NOTES

The typical use of this service is within user space device drivers. When a device interrupt occurs, the (in this case, simple) kernel device driver would place device status in a shared (with the user device driver) memory window (established by **device_map**) and signal the associated event. The user space device driver would normally be waiting with **evc_wait**. The user thread then wakes, processes the device status, typically interacting with the device via its shared memory window, then waits for the next interrupt.

RETURN VALUE

KERN_SUCCESS
The event has occurred.

KERN_INVALID_ARGUMENT
The event object is damaged.

KERN_NO_SPACE
There is already a thread waiting for this event.

RELATED INFORMATION

Functions: **device_map**.

exception_raise

Function — Sends an exception message

LIBRARY

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

SYNOPSIS

```
kern_return_t exception_raise
                (mach_port_t          exception_port,
                 mach_port_t          thread,
                 mach_port_t          task,
                 int                  exception,
                 int                  code,
                 int                  subcode);
```

DESCRIPTION

The **exception_raise** function can be used to send an exception message to an exception server. This function is normally called only by a thread in the context of the kernel when it takes an exception. It may be called by intermediaries to signal an exception to an exception server. Note that calling this function does not cause the specified thread to take an exception; it is called to signify that the thread did take the specified exception.

PARAMETERS

exception_port
: [in scalar] The port to which the exception notification is to be sent. This is normally the port named via **thread_set_special_port** or **task_set_special_port**.

thread
: [in scalar] The control port to the thread taking the exception.

task
: [in scalar] The control port to 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 **kern_return_t** describing error. *subcode* contains 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* and *subcode* fields.

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

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

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

subcode
: [in scalar] A specific type of *code*.

RETURN VALUE

KERN_SUCCESS
: The exception server has indicated that the thread is to continue from the point of exception.

Other values indicate that the exception was handled directly and the thread was restarted or terminated by the exception handler.

RELATED INFORMATION

Functions: **catch_exception_raise**, **exc_server.**

mach_sample_thread

Function — Perform periodic PC sampling for a thread

SYNOPSIS

```
kern_return_t mach_sample_thread
                (mach_port_t                    task,
                mach_port_t                     reply_port,
                mach_port_t                     sample_thread);
```

DESCRIPTION

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

PARAMETERS

task
: [in scalar] Random task port on the same node as *sample_thread*. (not used)

reply_port
: [in scalar] Port to which PC sample buffers are sent. A value of MACH_PORT_NULL stops PC sampling for the thread.

sample_thread
: [in scalar] Thread whose PC is to be sampled

NOTES

Once PC sampling (profiling) is enabled for a thread, the kernel will, at random times, send a buffer full of PC samples to the specified *reply_port*. These buffers have the following format:

```
struct message
{
        mach_msg_header_t          head;
        mach_msg_type_t            type;
        int                        arg [SIZE_PROF_BUFFER+1];
};
```

The message ID is 666666. (SIZE_PROF_BUFFER is defined in **mach/profilparam.h**). *arg* [SIZE_PROF_BUFFER] specifies the number of values actually sent. If this value is less than SIZE_PROF_BUFFER, it means that this is the last buffer to be sent (PC sampling had been turned off for the thread).

RETURN VALUE

KERN_SUCCESS
: PC sampling has been enabled/disabled.

KERN_INVALID_ARGUMENT
: *task*, *reply_port*, or *sample_thread* are not valid

KERN_RESOURCE_SHORTAGE
: Some critical kernel resource is unavailable.

RELATED INFORMATION

Functions: **mach_sample_task**.

mach_thread_self

System Trap — Returns the thread self port

LIBRARY

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

SYNOPSIS

mach_port_t **mach_thread_self**
();

DESCRIPTION

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

PARAMETERS

None

RETURN VALUE

Send rights to the thread's port.

RELATED INFORMATION

Functions: **thread_info**.

swtch

System Trap — Attempt a context switch

LIBRARY

Not declared anywhere.

SYNOPSIS

boolean_t **swtch**
();

DESCRIPTION

The **swtch** function attempts to context switch the current thread off the processor.

This function is useful in user level lock management routines. If the current thread cannot make progress because of some lock, it would execute the **swtch** function. When this returns, the thread should once again try to make progress by attempting to obtain its lock.

This function returns a flag indicating whether there is anything else for the processor to do. If there is nothing else, the thread can spin waiting for its lock, instead of continuing to call **swtch**.

PARAMETERS

None

RETURN VALUE

TRUE
: There are other threads that the processor could run.

FALSE
: The processor has nothing better to do.

RELATED INFORMATION

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

swtch_pri

System Trap — Attempt a context switch to low priority

LIBRARY

Not declared anywhere.

SYNOPSIS

boolean_t **swtch_pri**
(int *priority*);

DESCRIPTION

The **swtch_pri** function attempts to context switch the current thread off the processor. The thread's priority is lowered to the minimum possible value during this time. The priority of the thread will be restored when it is awakened.

This function is useful in user level lock management routines. If the current thread cannot make progress because of some lock, it would execute the **swtch_pri** function. When this returns, the thread should once again try to make progress by attempting to obtain its lock.

This function returns a flag indicating whether there is anything else for the processor to do. If there is nothing else, the thread can spin waiting for its lock, instead of continuing to call **swtch_pri**.

PARAMETERS

priority
[in scalar] Currently not used.

RETURN VALUE

TRUE
There are other threads that the processor could run.

FALSE
The processor has nothing better to do.

RELATED INFORMATION

Functions: **swtch**, **thread_abort**, **thread_depress_abort**, **thread_switch**.

thread_abort

Function — Aborts a thread

SYNOPSIS

```
kern_return_t thread_abort
        (mach_port_t                    target_thread);
```

DESCRIPTION

The **thread_abort** function aborts page faults and any message primitive calls (**mach_msg**, **mach_msg_receive**, and **mach_msg_send**) in use by *target_thread*. (Note, though, that the message calls retry interrupted message operations unless MACH_SEND_INTERRUPT and MACH_RCV_INTERRUPT are specified.) Priority depressions are also aborted. The call returns a code indicating that it was interrupted. The call is interrupted even if the thread (or the task containing it) is suspended. If it is suspended, the thread receives the interrupt when it resumes.

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

The basic purpose of **thread_abort** is to let one thread cleanly stop another thread (*target_thread*). The target thread is stopped in such a manner that its future execution can be controlled in a predictable way.

PARAMETERS

target_thread
: [in scalar] The thread to be aborted.

NOTES

By way of comparison, the **thread_suspend** function keeps the target thread from executing any further instructions at the user level, including the return from a system call. The **thread_get_state** function returns the thread's user state, while **thread_set_state** allows modification of the user state.

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

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

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

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

CAUTIONS

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

RETURN VALUE

KERN_SUCCESS
: The thread received an interrupt.

KERN_INVALID_ARGUMENT
: *target_thread* is not a valid thread.

RELATED INFORMATION

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

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

PARAMETERS

parent_task
: [in scalar] The task that is to contain the new thread.

child_thread
: [out scalar] 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.

RETURN VALUE

KERN_SUCCESS
: A new thread has been created.

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

KERN_RESOURCE_SHORTAGE
: Some critical kernel resource is unavailable.

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_depress_abort

Function — Cancel thread priority depression

SYNOPSIS

kern_return_t **thread_depress_abort**
(mach_port_t *thread*);

DESCRIPTION

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

PARAMETERS

thread
: [in scalar] Thread whose priority depression is canceled.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_ARGUMENT
: *thread* is not a valid thread.

RELATED INFORMATION

Functions: **swtch**, **swtch_pri**, **thread_abort**, **thread_switch**.

thread_get_special_port

Function — Returns a send right to a special port

SYNOPSIS

kern_return_t **thread_get_special_port**
(mach_port_t *thread,*
int *which_port,*
mach_port_t* *special_port*);

DESCRIPTION

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

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

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

MACRO FORMS

thread_get_exception_port

kern_return_t **thread_get_exception_port**
(mach_port_t *thread,*
mach_port_t* *special_port)*

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

thread_get_kernel_port

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

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

PARAMETERS

thread

[in scalar] 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_EXCEPTION_PORT
The thread's exception port. Used to receive exception messages from the kernel.

THREAD_KERNEL_PORT
The port used to name the thread. Used to invoke operations that affect the thread.

special_port
[out scalar] The returned value for the port.

RETURN VALUE

KERN_SUCCESS
The port was returned.

KERN_INVALID_ARGUMENT
thread is not a valid thread or *which_port* is not a valid port selector.

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,*
int	*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.

For *old_state*, the calling thread supplies an array of integers. On return, *old_state* contains the requested information.

For *old_stateCnt*, the calling thread specifies the maximum number of integers in *old_state*. On return, *old_stateCnt* contains the actual number of integers in *old_state*.

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

PARAMETERS

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

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

old_state
: [out array of *int*] Array of state information for the specified thread.

old_stateCnt
: [pointer to in/out scalar] The size of the state array. The maximum size is defined by THREAD_STATE_MAX.

RETURN VALUE

KERN_SUCCESS
: The state has been returned.

KERN_INVALID_ARGUMENT
: *target_thread* is not a valid thread, or specifies the calling thread, or *flavor* is not a valid type.

MIG_ARRAY_TOO_LARGE
: The returned array is too large for *old_state*. The function fills *old_state* and sets *old_stateCnt* to the number of elements that would have been returned if there had been enough space.

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,*
int	*flavor,*
thread_info_t	*thread_info,*
mach_msg_type_number_t*	*thread_infoCnt*);

DESCRIPTION

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

For *thread_info*, the calling thread supplies an array of integers. On return, *thread_info* contains the requested information.

For *thread_infoCnt*, the calling thread specifies the maximum number of integers in *thread_info*. On return, *thread_infoCnt* contains the actual number of integers in *thread_info*.

Currently, THREAD_BASIC_INFO and THREAD_SCHED_INFO are the only types of information supported. The size is defined by THREAD_BASIC_INFO_COUNT or THREAD_SCHED_INFO_COUNT, respectively.

PARAMETERS

target_thread
: [in scalar] 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.

  THREAD_SCHED_INFO
  : Returns scheduling information about the thread, such as priority and scheduling policy.

thread_info
: [out array of *int*] Information about the specified thread.

thread_infoCnt
[pointer to in/out scalar] The size of the information structure. The maximum size is defined by THREAD_INFO_MAX. Possible values are THREAD_BASIC_INFO_COUNT (for THREAD_BASIC_INFO) and THREAD_SCHED_INFO_COUNT (for THREAD_SCHED_INFO).

RETURN VALUE

KERN_SUCCESS
The thread information has been returned.

KERN_INVALID_ARGUMENT
target_thread is not a valid thread or *flavor* is not a valid type.

MIG_ARRAY_TOO_LARGE
The returned array is too large for *thread_info*. The function fills *thread_info* and sets *thread_infoCnt* to the number of elements that would have been returned if there had been enough space.

RELATED INFORMATION

Functions: **task_info**, **task_threads**, **thread_get_special_port**, **thread_get_state**, **thread_set_special_port**, **thread_set_state**.

Data Structures: **thread_basic_info**, **thread_sched_info**.

thread_max_priority

Function — Sets the maximum scheduling priority for a thread

LIBRARY

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

SYNOPSIS

```
kern_return_t thread_max_priority
                (mach_port_t                    thread,
                 mach_port_t                    processor_set,
                 int                            priority);
```

DESCRIPTION

The **thread_max_priority** function sets the maximum scheduling priority for *thread*.

Threads have three priorities associated with them by the system:

- A priority value which 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 which 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 which 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).

This function changes the maximum priority for the thread. Because this function requires the presentation of the corresponding processor set control port, this call can reset the maximum priority to any legal value.

PARAMETERS

thread
: [in scalar] The thread whose maximum scheduling priority is to be set.

processor_set
: [in scalar] The control port for the processor set to which the thread is currently assigned.

priority
: [in scalar] The new maximum priority for the thread.

RETURN VALUE

KERN_SUCCESS
: The priority has been set.

KERN_INVALID_ARGUMENT
: *thread* is not a valid thread, or *processor_set* does not name the processor set to which *thread* is currently assigned.

RELATED INFORMATION

Functions: **thread_priority, thread_policy, task_priority, processor_set_-max_priority**.

thread_policy

Function — Sets the scheduling policy to apply to a thread

LIBRARY

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

SYNOPSIS

```
kern_return_t thread_policy
        (mach_port_t                    thread,
         int                            policy,
         int                            data);
```

DESCRIPTION

The **thread_policy** function sets the scheduling policy to be applied to *thread*.

PARAMETERS

thread
: [in scalar] The thread scheduling policy is to be set.

policy
: [in scalar] Policy to be set. The values currently defined are POLICY_-TIMESHARE and POLICY_FIXEDPRI.

data
: [in scalar] Policy specific data. Currently, this value is used only for POLICY_FIXEDPRI, in which case it is the quantum to be used (in milliseconds); to be meaningful, this value must be a multiple of the basic system quantum (which can be obtained from **host_info**).

RETURN VALUE

KERN_SUCCESS
: The policy has been set.

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

KERN_FAILURE
: The processor set to which *thread* is currently assigned does not permit *policy.*

RELATED INFORMATION

Functions: **processor_set_policy_enable, processor_set_policy_disable**.

thread_priority

Function — Sets the scheduling priority for a thread

LIBRARY

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

SYNOPSIS

```
kern_return_t thread_priority
                (mach_port_t                    thread,
                 int                            priority,
                 boolean_t                      set_max);
```

DESCRIPTION

The **thread_priority** function sets the scheduling priority for *thread*.

PARAMETERS

thread
: [in scalar] The thread whose scheduling priority is to be set.

priority
: [in scalar] The new priority for the thread.

set_max
: [in scalar] True if the thread's maximum priority should also be set.

NOTES

Threads have three priorities associated with them by the system:

- A priority value which 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 which 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 which is sued 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).

This function changes the priority and optionally the maximum priority (if *set_-max* is TRUE) for *thread*. Priorities range from 0 to 31, where lower numbers denote higher priorities. If the new priority is higher than the priority of the current thread, preemption may occur as a result of this call. This call will fail if *priority* is greater than the current maximum priority of the thread. As a result, this call can only lower the value of a thread's maximum priority.

RETURN VALUE

KERN_SUCCESS
: The priority has been set.

KERN_INVALID_ARGUMENT
: *thread* is not a valid thread, or the priority value is out of range for priority values.

KERN_FAILURE
: The requested operation would violate the thread's maximum priority.

RELATED INFORMATION

Functions: **thread_max_priority, thread_policy, task_priority, processor_set_max_priority**.

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, you must repeat **thread_resume** until the count reaches zero.

PARAMETERS

target_thread
[in scalar] The thread to be resumed.

RETURN VALUE

KERN_SUCCESS
The thread's suspend count has been decremented.

KERN_FAILURE
The thread's suspend count is already at zero. A suspend count must be either zero or positive.

KERN_INVALID_ARGUMENT
target_thread is not a valid thread.

RELATED INFORMATION

Functions: **task_resume**, **task_suspend**, **thread_create**, **thread_info**, **thread_-suspend**, **thread_terminate**.

thread_set_special_port

Function — Sets a special port for a thread

SYNOPSIS

```
kern_return_t thread_set_special_port
        (mach_port_t        thread,
         int                which_port,
         mach_port_t        special_port);
```

DESCRIPTION

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

MACRO FORMS

thread_set_exception_port

```
kern_return_t thread_set_exception_port
        (mach_port_t        thread,
         mach_port_t        special_port)
```

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

thread_set_kernel_port

```
kern_return_t thread_set_kernel_port
        (mach_port_t        thread,
         mach_port_t        special_port)
```

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

PARAMETERS

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

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

THREAD_EXCEPTION_PORT
: The thread's exception port. Used to receive exception messages from the kernel.

THREAD_KERNEL_PORT
: The thread's kernel port. Used by the kernel to receive messages from the thread.

special_port
[in scalar] The value for the port.

RETURN VALUE

KERN_SUCCESS
The port was set.

KERN_INVALID_ARGUMENT
thread is not a valid thread or *which_port* is not a valid port selector.

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,
         int                         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*. *flavor* specifies the type of state to set.

For *new_state*, the calling thread supplies an array of integers.

For *new_stateCnt*, the calling thread specifies the maximum number of integers in *new_state*.

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

PARAMETERS

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

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

new_state
: [pointer to in array of *int*] Array of state information for the specified thread.

new_stateCnt
: [in scalar] The size of the state array. The maximum size is defined by THREAD_STATE_MAX.

RETURN VALUE

KERN_SUCCESS
: The state has been set.

KERN_INVALID_ARGUMENT
target_thread is not a valid thread, or specifies the calling thread, or *flavor* is not a valid type.

MIG_ARRAY_TOO_LARGE
The state array is too large for *new_state*. The function fills *new_state* and sets *new_stateCnt* to the number of elements that would have been returned if there had been enough space.

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 may continue to execute until it reaches the system return code or it may suspend within the kernel code. In either case, the system trap returns when the thread resumes.

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

PARAMETERS

target_thread
: [in scalar] The thread to be suspended.

CAUTIONS

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

RETURN VALUE

KERN_SUCCESS
: The thread has been suspended.

KERN_INVALID_ARGUMENT
: *target_thread* is not a valid thread.

RELATED INFORMATION

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

thread_switch

System Trap — Cause context switch with options

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t thread_switch
        (mach_port_t          new_thread,
         int                  option,
         int                  time);
```

DESCRIPTION

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

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

SWITCH_OPTION_NONE
: The *time* argument is ignored.

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

SWITCH_OPTION_DEPRESS
: The thread's priority is depressed to the lowest possible value for *time*. The priority depression is aborted when *time* has passed, when the current thread is next run (either via hand-off scheduling or because the processor set has nothing better to do), or when **thread_abort** or **thread_depress_abort** is applied to the current thread. Changing the thread's priority (via **thread_priority**) will not affect this depression.

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

PARAMETERS

new_thread
[in scalar] Thread to which the processor should switch context.

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

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

NOTES

thread_switch is often called when the current thread can proceed no further for some reason; the various options and arguments allow information about this reason to be transmitted to the kernel. The *new_thread* argument (hand-off scheduling) is useful when the identity of the thread that must make progress before the current thread runs again is known. The SWITCH_OPTION_WAIT option is used when the amount of time that the current thread must wait before it can do anything useful can be estimated and is fairly short, especially when the identity of the thread for which this thread must wait is not known.

CAUTIONS

Users should beware of calling **thread_switch** with an invalid hint (e.g., THREAD_NULL) and no option. Because the time-sharing scheduler varies the priority of threads based on usage, this may result in a waste of CPU time if the thread that must be run is of lower priority. The use of the SWITCH_OPTION_DEPRESS option in this situation is highly recommended.

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

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_ARGUMENT
new_thread is not a valid thread, or *option* is not a recognized option.

RELATED INFORMATION

Functions: **swtch**, **swtch_pri**, **thread_abort**, **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 kills creates *target_thread*.

PARAMETERS

target_thread
[in scalar] The thread to be destroyed.

RETURN VALUE

KERN_SUCCESS
The thread has been killed.

KERN_INVALID_ARGUMENT
target_thread is not a valid thread.

RELATED INFORMATION

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

thread_wire

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

LIBRARY

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

SYNOPSIS

```
kern_return_t thread_wire
                (mach_port_t                    host_priv,
                 mach_port_t                    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 scalar] The privileged control port for the host on which the thread executes.

thread
: [in scalar] The thread to be wired.

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

RETURN VALUE

KERN_SUCCESS
: The thread is wired.

KERN_INVALID_ARGUMENT
: *thread* is not a valid thread or *host_priv* is not the control port for the host on which *thread* executes.

RELATED INFORMATION

Functions: **vm_wire**.

CHAPTER 7 Task Interface

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

mach_sample_task

Function — Perform periodic PC sampling for a task

SYNOPSIS

```
kern_return_t mach_sample_task
                (mach_port_t                    task,
                mach_port_t                     reply_port,
                mach_port_t                     sample_task);
```

DESCRIPTION

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

PARAMETERS

task
: [in scalar] Random task port on the same node as *sample_task*. (not used)

reply_port
: [in scalar] Port to which PC sample buffers are sent. A value of MACH_PORT_NULL stops PC sampling for the task.

sample_task
: [in scalar] Task whose threads' PC are to be sampled

NOTES

Once PC sampling (profiling) is enabled for a task, the kernel will, at random times, send a buffer full of PC samples to the specified *reply_port*. These buffers have the following format:

```
[1]  struct message
[2]  {
[3]          mach_msg_header_t          head;
[4]          mach_msg_type_t            type;
[5]          int                        arg [SIZE_PROF_BUFFER+1];
[6]  };
```

The message ID is 666666. (SIZE_PROF_BUFFER is defined in **mach/profil-param.h**). *arg* [SIZE_PROF_BUFFER] specifies the number of values actually sent. If this value is less than SIZE_PROF_BUFFER, it means that this is the last buffer to be sent (PC sampling had been turned off for the task).

RETURN VALUE

KERN_SUCCESS
PC sampling has been enabled/disabled.

KERN_INVALID_ARGUMENT
task, *reply_port*, or *sample_task* are not valid

KERN_RESOURCE_SHORTAGE
Some critical kernel resource is unavailable.

RELATED INFORMATION

Functions: **mach_sample_thread**.

mach_task_self

System Trap — Returns the task self port

LIBRARY

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

SYNOPSIS

```
mach_port_t mach_task_self
                ();
```

DESCRIPTION

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

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.

PARAMETERS

None

RETURN VALUE

Send rights to the task's port.

RELATED INFORMATION

Functions: **task_info**.

task_create

Function — Creates a task

SYNOPSIS

```
kern_return_t task_create
        (mach_port_t        parent_task,
         boolean_t          inherit_memory,
         mach_port_t*       child_task);
```

DESCRIPTION

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

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

- **task_kernel_port** — The port by which the kernel knows the new child task. The child task holds a send right for this port. The port name is also returned to the calling task.
- **task_bootstrap_port** — The port to which the child task can send a message requesting return of any system service ports that it needs (for example, a port to the Network Name Server or the Environment Manager). The child task inherits a send right for this port from the parent task. The child task can use **task_get_special_port** to change this port.
- **task_exception_port** — A default exception port for the child task, inherited from the parent task. The exception port is the port to which the kernel sends exception messages. Exceptions are synchronous interruptions to the normal flow of program control caused by the program itself. Some exceptions are handled transparently by the kernel, but others must be reported to the program. The child task, or any one of its threads, can change the default exception port to take an active role in exception handling (see **task_get_special_port** or **thread_get_special_port**).

The child task inherits the PC sampling state of the parent.

PARAMETERS

parent_task
: [in scalar] The task from which to draw the child task's port rights, resource limits, and address space.

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

child_task
[out scalar] The kernel-assigned name for the new task.

RETURN VALUE

KERN_SUCCESS
A new task has been created.

KERN_INVALID_ARGUMENT
parent_task is not a valid task port.

KERN_RESOURCE_SHORTAGE
Some critical kernel resource is unavailable.

RELATED INFORMATION

Functions: **task_get_special_port**, **task_resume**, **task_set_special_port**, **task_-suspend**, **task_terminate**, **task_threads**, **thread_create**, **thread_resume**, **vm_inherit**, **mach_sample_task**.

task_get_emulation_vector

Function — Return user-level handlers for system calls.

SYNOPSIS

kern_return_t **task_get_emulation_vector**

(mach_port_t	*task,*
int*	*vector_start*,
emulation_vector_t*	*emulation_vector,*
mach_msg_type_number_t*	*emulation_vector_count*);

DESCRIPTION

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

PARAMETERS

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

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

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

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

RETURN VALUE

KERN_SUCCESS
: The emulation handler addresses were returned.

EML_BAD_TASK
: *task* is not a valid task.

RELATED INFORMATION

Functions: **task_set_emulation**, **task_set_emulation_vector**.

task_get_special_port

Function — Returns a send right to a special port

SYNOPSIS

kern_return_t **task_get_special_port**
(mach_port_t *task,*
int *which_port,*
mach_port_t* *special_port*);

DESCRIPTION

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

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

MACRO FORMS

task_get_bootstrap_port

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

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

task_get_exception_port

kern_return_t **task_get_exception_port**
(mach_port_t *task,*
mach_port_t* *special_port)*

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

task_get_kernel_port

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

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

PARAMETERS

task
[in scalar] 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
The port used to name this task. Used to send messages that affect the task.

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

TASK_EXCEPTION_PORT
The task's exception port. Used to receive exception messages from the kernel.

special_port
[out scalar] The returned value for the port.

RETURN VALUE

KERN_SUCCESS
The port was returned.

KERN_INVALID_ARGUMENT
task is not a valid task or *which_port* is not a valid port selector.

RELATED INFORMATION

Functions: **mach_task_self**, **task_create**, **task_set_special_port**, **thread_-get_special_port**, **thread_set_special_port**.

task_info

Function — Returns information about a task

SYNOPSIS

kern_return_t **task_info**

(mach_port_t	*target_task,*
int	*flavor,*
task_info_t	*task_info,*
mach_msg_type_number_t*	*task_infoCnt*);

DESCRIPTION

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

For *task_info*, the calling task or thread supplies an array of integers. On return, *task_info* contains the requested information.

For *task_infoCnt*, the calling task or thread specifies the maximum number of integers in *task_info*. On return, *task_infoCnt* contains the actual number of integers in *task_info*.

Currently, TASK_BASIC_INFO and TASK_THREAD_TIMES_INFO are the only types of information supported. Their sizes are defined by TASK_BASIC_INFO_COUNT and TASK_THREAD_TIMES_INFO_COUNT, respectively.

PARAMETERS

target_task
[in scalar] 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.

TASK_THREAD_TIMES_INFO
Returns system and user space run-times for live threads.

task_info
[out array of *int*] Information about the specified task.

task_infoCnt
[pointer to in/out scalar] The size of the information structure. The maximum size is defined by TASK_INFO_MAX. Currently, the only valid values are TASK_BASIC_INFO_COUNT (for TASK_BASIC_INFO) and TASK_THREAD_TIMES_INFO_COUNT (for TASK_THREAD_TIMES_INFO).

RETURN VALUE

KERN_SUCCESS
The task information has been returned.

KERN_INVALID_ARGUMENT
target_task is not a valid task or *flavor* is not a valid type.

MIG_ARRAY_TOO_LARGE
The returned array is too large for *task_info*. The function fills *task_info* and sets *task_infoCnt* to the number of elements that would have been returned if there had been enough space.

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**, **task_thread_times_info**.

task_priority

Function — Sets the scheduling priority for a task

LIBRARY

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

SYNOPSIS

```
kern_return_t task_priority
              (mach_port_t                    task,
               int                            priority,
               boolean_t                      change_threads);
```

DESCRIPTION

The **task_priority** function sets the scheduling priority for *task*. The priority of a task is used only when creating new threads. A new thread's priority is set to that of the enclosing task's priority. Changing the priority of a task does not affect the priority of the enclosed threads unless *change_threads* is TRUE. If this priority change violates the maximum priority of some threads, as many threads as possible will be changed and an error code will be returned.

PARAMETERS

task
: [in scalar] The task whose scheduling priority is to be set.

priority
: [in scalar] The new priority for the task.

change_threads
: [in scalar] True if priority of existing threads within the task should also be changed.

RETURN VALUE

KERN_SUCCESS
: The priority has been set.

KERN_INVALID_ARGUMENT
: *task* is not a valid task, or the priority value is out of range for priority values.

KERN_FAILURE

change_threads was TRUE and the attempt to change the priority of some existing thread within the task failed because the new priority would violate that thread's maximum priority.

RELATED INFORMATION

Functions: **thread_max_priority, thread_priority, processor_set_max_priority**.

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 scalar] The task to be resumed.

RETURN VALUE

KERN_SUCCESS
: The task's suspend count has been decremented.

KERN_FAILURE
: The task's suspend count is already at zero. A suspend count must be either zero or positive.

KERN_INVALID_ARGUMENT
: *target_task* is not a valid task.

RELATED INFORMATION

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

task_set_emulation

Function — Establish a user-level handler for a system call.

SYNOPSIS

kern_return_t **task_set_emulation**

(mach_port_t	*task,*
vm_address_t	*routine_entry_pt,*
int	*syscall_number*);

DESCRIPTION

The **task_set_emulation** function establishes a handler within the task for a particular system call. When a thread executes a system call with this particular number, the system call will be redirected to the specified routine within the task's address space. This is expected to be an address within the transparent emulation library.

These emulation handler addresses are inherited by child processes.

PARAMETERS

task
: [in scalar] The task for which to establish the system call handler.

routine_entry_pt
: [in scalar] The address within the task of the handler for this particular system call.

syscall_number
: [in scalar] The number of the system call to be handled by this handler.

RETURN VALUE

KERN_SUCCESS
: The emulation handler was set.

EML_BAD_TASK
: *task* is not a valid task.

EML_BAD_CNT
: *syscall_number* is not an allowed system call number.

RELATED INFORMATION

Functions: **task_set_emulation_vector**, **task_get_emulation_vector**.

task_set_emulation_vector

Function — Establishes user-level handlers for system calls.

SYNOPSIS

```
kern_return_t task_set_emulation_vector
        (mach_port_t                    task,
         int                            vector_start,
         emulation_vector_t             emulation_vector,
         mach_msg_type_number_t         emulation_vector_count);
```

DESCRIPTION

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

These emulation handler addresses are inherited by child processes.

PARAMETERS

task
: [in scalar] The task for which to establish the system call handler.

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

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

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

RETURN VALUE

KERN_SUCCESS
: The emulation handler was set.

EML_BAD_TASK
: *task* is not a valid task.

EML_BAD_CNT
An element of the vector had a syscall number out of range.

RELATED INFORMATION

Functions: **task_set_emulation**, **task_get_emulation_vector**.

task_set_special_port

Function — Sets a special port for a task

SYNOPSIS

```
kern_return_t task_set_special_port
            (mach_port_t                      task,
             int                              which_port,
             mach_port_t                      special_port);
```

DESCRIPTION

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

MACRO FORMS

task_set_bootstrap_port

```
kern_return_t task_set_bootstrap_port
            (mach_port_t                      task,
             mach_port_t                      special_port)
```

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

task_set_exception_port

```
kern_return_t task_set_exception_port
            (mach_port_t                      task,
             mach_port_t                      special_port)
```

⇒ **task_set_special_port** (*task*, TASK_EXCEPTION_PORT, *special_port*).

task_set_kernel_port

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

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

PARAMETERS

task
: [in scalar] The task for which to set the port.

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

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

TASK_EXCEPTION_PORT
The task's exception port. Used to receive exception messages from the kernel.

TASK_KERNEL_PORT
The task's kernel port. Used by the kernel to receive messages from the task.

special_port
[in scalar] The value for the port.

RETURN VALUE

KERN_SUCCESS
The port was set.

KERN_INVALID_ARGUMENT
task is not a valid task or *which_port* is not a valid port selector.

RELATED INFORMATION

Functions: **task_create**, **task_get_special_port**, **exception_raise**, **mach_task_-self**, **thread_get_special_port**, **thread_set_special_port**.

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.

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

PARAMETERS

target_task
[in scalar] The task to be suspended.

RETURN VALUE

KERN_SUCCESS
The task has been suspended.

KERN_INVALID_ARGUMENT
target_task is not a valid task.

RELATED INFORMATION

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

task_terminate

Function — Destroys a task

SYNOPSIS

```
kern_return_t task_terminate
                (mach_port_t                target_task);
```

DESCRIPTION

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

PARAMETERS

target_task
: [in scalar] The task to be destroyed.

RETURN VALUE

KERN_SUCCESS
: The task has been killed.

KERN_INVALID_ARGUMENT
: *target_task* is not a valid task.

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,
                 thread_array_t*                   thread_list,
                 mach_msg_type_number_t*           thread_count);
```

DESCRIPTION

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

PARAMETERS

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

thread_list
: [out pointer to dynamic array of *thread_t*] The returned list of threads within *target_task*, in no particular order.

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

RETURN VALUE

KERN_SUCCESS
: The list of threads has been returned.

KERN_INVALID_ARGUMENT
: *target_task* is not a valid task.

RELATED INFORMATION

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

CHAPTER 8 Host Interface

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

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

host_adjust_time

Function —Gradually change the time

LIBRARY

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

SYNOPSIS

```
kern_return_t host_adjust_time
                (mach_port_t                 host_priv,
                 time_value_t           new_adjustment,
                 time_value_t*          old_adjustment);
```

DESCRIPTION

The **host_adjust_time** function arranges for the time on a specified host to be gradually changed by an adjustment value.

PARAMETERS

host_priv
: [in scalar] The control port the host for which the time is to be set.

new_adjustment
: [in structure] New adjustment value.

old_adjustment
: [out structure] Old adjustment value.

RETURN VALUE

KERN_SUCCESS
: The time is being adjusted.

KERN_INVALID_HOST
: The supplied host port is not the privileged host port.

RELATED INFORMATION

Functions: **host_get_time**, **host_set_time**.

Data Structures: **time_value**.

host_get_boot_info

Function — Return operator boot information

LIBRARY

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

priv_host
: [in scalar] The control port for the host for which information is to be obtained.

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

RETURN VALUE

KERN_SUCCESS
: The information has been returned.

KERN_INVALID_ARGUMENT
: *priv_host* is not a host control port.

KERN_INVALID_ADDRESS
: *version* points to inaccessible memory.

RELATED INFORMATION

Functions: **host_info**.

host_get_time

Function —Return the current time.

LIBRARY

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

SYNOPSIS

```
kern_return_t host_get_time
                (mach_port_t                    host,
                 time_value_t*                  current_time);
```

DESCRIPTION

The **host_get_time** function returns the current time as seen by that host.

PARAMETERS

host
: [in scalar] The name port the host for which the time is to be set.

current_time
: [out structure] Returned time value.

RETURN VALUE

KERN_SUCCESS
: The current time is returned.

RELATED INFORMATION

Functions: **host_adjust_time**, **host_set_time**.

Data Structures: **time_value**.

host_info

Function — Returns information about a host

LIBRARY

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

SYNOPSIS

```
kern_return_t host_info
        (mach_port_t                    host,
         int                            flavor,
         host_info_t                    host_info,
         mach_msg_type_number_t*        host_infoCnt);
```

DESCRIPTION

The **host_info** function returns selected information about a host, as specified by *flavor*. *host_info* is an array of integers that is supplied by the caller, and filled with the specified information. *host_infoCnt* is supplied as the maximum number of integers in *host_info*. On return, it contains the actual number of integers in *host_info*.

Basic information is defined by HOST_BASIC_INFO. Processor slots of the active (available) processors are defined by HOST_PROCESSOR_SLOTS. Additional information of interest to schedulers is defined by HOST_LOAD_INFO and HOST_SCHED_INFO.

PARAMETERS

host
: [in scalar] The name port for the host for which information is to be obtained.

flavor
: [in scalar] The type of statistics desired. Currently, HOST_BASIC_INFO, HOST_LOAD_INFO, HOST_PROCESSOR_SLOTS and HOST_SCHED_INFO are defined.

host_info
: [out array of *int*] Statistics about the specified host. The relevant structures are **host_basic_info, host_load_info** and **host_sched_info**. In the case of HOST_PROCESSOR_SLOTS, the return value is an array of processor slot numbers for active processors.

host_infoCnt
[pointer to in/out scalar] Size of the information structure, in units of sizeof(int). This should be HOST_BASIC_INFO_COUNT (for HOST_BASIC_INFO), HOST_SCHED_INFO_COUNT (for HOST_SCHED_INFO), HOST_LOAD_INFO_COUNT (for HOST_-LOAD_INFO) and the maximum number of CPUs reported by HOST_BASIC_INFO (for HOST_PROCESSOR_SLOTS).

RETURN VALUE

KERN_SUCCESS
The information has been returned.

KERN_INVALID_ARGUMENT
host is not a host port or *flavor* is not recognized.

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

RELATED INFORMATION

Functions: **host_get_boot_info**, **host_kernel_version, host_processors, processor_info**.

Data Structures: **host_basic_info, host_load_info, host_sched_info**

host_kernel_version

Function — Returns kernel version information for a host

LIBRARY

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

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 scalar] The name port for the host for which information is to be obtained.

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

RETURN VALUE

KERN_SUCCESS
: The information has been returned.

KERN_INVALID_ARGUMENT
: *host* is not a host port.

KERN_INVALID_ADDRESS
: *version* points to inaccessible memory.

RELATED INFORMATION

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

host_reboot

Function — Reboot this host

LIBRARY

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

SYNOPSIS

```
kern_return_t host_reboot
                (mach_port_t                    host_priv,
                 int                            options);
```

DESCRIPTION

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

PARAMETERS

host_priv
: [in scalar] 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

KERN_NO_ACCESS
: The supplied host port is not the privileged host port.

host_set_time

Function — Sets the time

LIBRARY

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

SYNOPSIS

```
kern_return_t host_set_time
                (mach_port_t          host_priv,
                 time_value_t         new_time);
```

DESCRIPTION

The **host_set_time** function establishes the time on the specified host.

PARAMETERS

host_priv
[in scalar] The control port for the host for which the time is to be set.

new_time
[in structure] Time to be set.

RETURN VALUE

KERN_SUCCESS
The time is set.

KERN_NO_ACCESS
The supplied host port is not the privileged host port.

RELATED INFORMATION

Functions: **host_adjust_time**, **host_get_time**.

Data Structures: **time_value**.

mach_host_self

System Trap — Returns the host self port

LIBRARY

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

SYNOPSIS

```
mach_port_t mach_host_self
                ();
```

DESCRIPTION

The **mach_host_self** function returns send rights to the current host's name port.

PARAMETERS

None

RETURN VALUE

Send rights to the host's name port.

RELATED INFORMATION

Functions: **host_info**.

CHAPTER 9 Processor Interface

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

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

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

host_processor_set_priv

Function — Returns a processor set control port for a host

LIBRARY

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

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 scalar] The control port for the host for which the processor set is desired.

set_name
: [in scalar] The name port for the processor set desired.

processor_set
: [out scalar] The returned processor set control port.

RETURN VALUE

KERN_SUCCESS
: The port has been returned.

KERN_INVALID_ARGUMENT
: *host_priv* is not a valid host control port.

RELATED INFORMATION

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

host_processor_sets

Function — Returns processor set ports for a host

LIBRARY

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

SYNOPSIS

kern_return_t **host_processor_sets**

(mach_port_t	*host,*
processor_set_name_array_t*	*processor_set_list,*
mach_msg_type_number_t*	*processor_set_count*);

DESCRIPTION

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

PARAMETERS

host
: [in scalar] The name port for the host for which the processor sets are desired.

processor_set_list
: [out pointer to dynamic array of *processor_set_name_t*] The set of processor set name ports for those currently existing on *host*; no particular order is guaranteed.

processor_set_count
: [out scalar] The number of processor sets returned.

NOTES

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

processor_set_list is automatically allocated by the kernel, as if by **vm_allocate**. It is good practice to **vm_deallocate** this space when it is no longer needed.

RETURN VALUE

KERN_SUCCESS
: The ports have been returned.

KERN_INVALID_ARGUMENT
host is not a valid host.

RELATED INFORMATION

Functions: **host_processor_set_priv, processor_set_create, processor_set_-tasks, processor_set_threads**.

host_processors

Function — Gets processor ports for a host

LIBRARY

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

SYNOPSIS

```
kern_return_t host_processors
              (mach_port_t                     host_priv,
               processor_array_t*              processor_list,
               mach_msg_type_number_t*         processor_count);
```

DESCRIPTION

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

PARAMETERS

host_priv
: [in scalar] The control port for the desired host.

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

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

RETURN VALUE

KERN_SUCCESS
: The list of ports is returned.

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

KERN_INVALID_ADDRESS
: *processor_count* points to invalid memory.

RELATED INFORMATION

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

processor_assign

Function — Assign a processor to a processor set

LIBRARY

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

SYNOPSIS

```
kern_return_t processor_assign
                (mach_port_t        processor,
                 mach_port_t        new_set,
                 boolean_t          wait);
```

DESCRIPTION

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

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

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

PARAMETERS

processor
: [in scalar] The processor to be assigned.

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

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

CAUTIONS

Intermediaries that interpose on ports must be sure to interpose on both ports involved in the call if they interpose on either.

RETURN VALUE

KERN_SUCCESS
The assignment was performed.

KERN_INVALID_ARGUMENT
processor is not a processor port, or *new_set* is not a processor set port for the same host as *processor.*

RELATED INFORMATION

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

processor_control

Function — Do something to a processor

LIBRARY

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

processor
: [in scalar] The processor to be controlled.

cmd
: [pointer to in array of *int*] An array containing the command to be applied to the processor.

count
: [in scalar] The size of the *cmd* array.

NOTES

These operations are machine dependent. They may do nothing.

RETURN VALUE

KERN_SUCCESS
: The operation was performed.

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

KERN_INVALID_ARGUMENT
processor is not a processor port.

KERN_INVALID_ADDRESS
cmd points to inaccessible memory.

RELATED INFORMATION

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

processor_exit

Function — Exit a processor

LIBRARY

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

SYNOPSIS

kern_return_t **processor_exit**
(mach_port_t *processor*);

DESCRIPTION

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

PARAMETERS

processor
[in scalar] The processor to be controlled.

NOTES

This operation is machine dependent. It may do nothing.

CAUTIONS

The ability to restart an exited processor is machine dependent.

RETURN VALUE

KERN_SUCCESS
The operation was performed.

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

KERN_INVALID_ARGUMENT
processor is not a processor port.

RELATED INFORMATION

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

processor_get_assignment

Function — Get current assignment for a processor

LIBRARY

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

SYNOPSIS

```
kern_return_t processor_get_assignment
                (mach_port_t          processor,
                 mach_port_t*         assigned_set);
```

DESCRIPTION

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

PARAMETERS

processor
: [in scalar] The processor whose assignment is desired.

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

RETURN VALUE

KERN_SUCCESS
: The processor set name was returned.

KERN_INVALID_ARGUMENT
: *processor* is not a processor port.

KERN_INVALID_ADDRESS
: *assigned_set* points to inaccessible memory.

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

RELATED INFORMATION

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

processor_info

Function — Returns information about a processor.

LIBRARY

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

SYNOPSIS

kern_return_t **processor_info**	
(mach_port_t	*processor,*
int	*flavor,*
mach_port_t*	*host,*
processor_info_t	*processor_info,*
mach_msg_type_number_t*	*processor_infoCnt*);

DESCRIPTION

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

Basic information is defined by PROCESSOR_BASIC_INFO. Additional information is defined by machine-dependent values of *flavor*.

PARAMETERS

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

flavor
: [in scalar] The type of information requested. Currently, only PROCESSOR_BASIC_INFO is defined.

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

processor_info
: [out array of *int*] Information about the processor.

processor_infoCnt
: [pointer to in/out scalar] Size of the info structure, in units of sizeof(int). This should be PROCESSOR_BASIC_INFO_COUNT (for PROCESSOR_BASIC_INFO).

RETURN VALUE

KERN_SUCCESS
The information has been returned.

KERN_INVALID_ARGUMENT
processor is not a processor port, or *flavor* is not recognized.

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

RELATED INFORMATION

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

Data Structures: **processor_basic_info.**

processor_set_create

Function — Creates a new processor set

LIBRARY

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

SYNOPSIS

```
kern_return_t processor_set_create
                (mach_port_t                host,
                 mach_port_t*               new_set,
                 mach_port_t*               new_name);
```

DESCRIPTION

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

PARAMETERS

host
: [in scalar] The name port for the host on which the set is to be created.

new_set
: [out scalar] Control port used for performing operations on the new set.

new_name
: [out scalar] Name port used to identify the new set and obtain information about it.

RETURN VALUE

KERN_SUCCESS
: The set was created.

KERN_INVALID_ARGUMENT
: *host* is not a host port.

KERN_INVALID_ADDRESS
: *new_set* and/or *new_name* point to inaccessible memory.

RELATED INFORMATION

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

processor_set_default

Function — Returns the default processor set

LIBRARY

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

SYNOPSIS

kern_return_t **processor_set_default**
(mach_port_t *host,*
mach_port_t* *default_set*);

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. The port returned can be used to obtain information about this set (such as how many threads are assigned to it). It cannot be used to perform operations on the set.

PARAMETERS

host
[in scalar] The name port for the host for which the default processor set is desired.

default_set
[out scalar] The returned name port for the default processor set.

RETURN VALUE

KERN_SUCCESS
The default set has been returned.

KERN_INVALID_ARGUMENT
host was not a host.

KERN_INVALID_ADDRESS
default_set points to inaccessible memory.

RELATED INFORMATION

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

processor_set_destroy

Function — Destroys a processor set

LIBRARY

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

SYNOPSIS

```
kern_return_t processor_set_destroy
        (mach_port_t                    processor_set);
```

DESCRIPTION

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

PARAMETERS

processor_set
: [in scalar] The control port for the processor set to be destroyed.

RETURN VALUE

KERN_SUCCESS
: The set was destroyed.

KERN_FAILURE
: An attempt was made to destroy the default processor set.

KERN_INVALID_ARGUMENT
: *processor_set* is not a processor set control port.

RELATED INFORMATION

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

processor_set_info

Function — Returns information about a processor set.

LIBRARY

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

SYNOPSIS

kern_return_t **processor_set_info**	
(mach_port_t	*processor_set,*
int	*flavor,*
mach_port_t*	*host,*
processor_set_info_t	*processor_set_info,*
mach_msg_type_number_t*	*infoCnt*);

DESCRIPTION

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

Basic information is defined by PROCESSOR_SET_BASIC_INFO. Scheduling information is given by PROCESSOR_SET_SCHED_INFO.

PARAMETERS

processor_set
: [in scalar] A processor set name or control port for which information is desired.

flavor
: [in scalar] The type of information requested. Currently, PROCESSOR_SET_BASIC_INFO and PROCESSOR_SET_SCHED_INFO are defined.

host
: [out scalar] The name port for the host on which the processor resides.

processor_set_info
: [out array of *int*] Information about the processor set.

infoCnt
: [pointer to in/out scalar] Size of the info structure, in units of sizeof(int). This should be PROCESSOR_SET_BASIC_INFO_-

COUNT (for PROCESSOR_SET_BASIC_INFO) and PROCESSOR_-SET_SCHED_INFO_COUNT (for PROCESSOR_SCHED_INFO).

RETURN VALUE

KERN_SUCCESS
The information has been returned.

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

MIG_ARRAY_TOO_LARGE
Returned info array is too large for *processor_set_info*. *processor_set_-info* is filled as much as possible. *infoCnt* is set to the number of elements that would be returned if there were enough room.

RELATED INFORMATION

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

Data Structures: **processor_set_basic_info**, **processor_set_sched_info.**

processor_set_max_priority

Function — Sets the maximum scheduling priority for a processor set

LIBRARY

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

SYNOPSIS

kern_return_t **processor_set_max_priority**

(mach_port_t	*processor_set,*
int	*priority,*
boolean_t	*change_threads*);

DESCRIPTION

The **processor_set_max_priority** function sets the maximum scheduling priority for *processor_set*. The maximum priority of a processor set is used only when creating new threads. A new thread's maximum priority is set to that of its assigned processor set. When assigned to a processor set, a thread's maximum priority is reduced, if necessary, to that of its new processor set; its current priority is also reduced, as needed. Changing the maximum priority of a processor set does not affect the priority of the currently assigned threads unless *change_threads* is TRUE. If this priority change violates the maximum priority of some threads, their maximum priorities will be reduced to match.

PARAMETERS

processor_set
: [in scalar] The control port for the processor set whose maximum scheduling priority is to be set.

priority
: [in scalar] The new priority for the processor set.

change_threads
: [in scalar] True if the maximum priority of existing threads assigned to this processor set should also be changed.

RETURN VALUE

KERN_SUCCESS
: The priority has been set.

KERN_INVALID_ARGUMENT
processor_set is not a valid processor set, or the priority value is out of range for priority values.

RELATED INFORMATION

Functions: **thread_max_priority, thread_priority, thread_assign**.

processor_set_policy_disable

Function — Disables a scheduling policy for a processor set

LIBRARY

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

SYNOPSIS

kern_return_t **processor_set_policy_disable**
(mach_port_t *processor_set,*
int *policy,*
boolean_t *change_threads*);

DESCRIPTION

The **processor_set_policy_disable** function restricts the set of scheduling policies allowed for *processor_set*. The set of scheduling policies allowed for a processor set is the set of policies allowed to be set for threads assigned to that processor set. The current set of permitted policies can be obtained from **processor_set_info**. Timesharing may not be forbidden for any processor set. This is a compromise to reduce the complexity of the assign operation; any thread whose policy is forbidden by its target processor set has its policy reset to timesharing. Disabling a scheduling policy for a processor set has no effect on threads currently assigned to that processor set unless *change_threads* is TRUE, in which case their policies will be reset to timesharing.

PARAMETERS

processor_set
: [in scalar] The control port for the processor set for which a scheduling policy is to be disabled.

policy
: [in scalar] Policy to be disabled. The values currently defined are POLICY_TIMESHARE and POLICY_FIXEDPRI.

change_threads
: [in scalar] If true, causes the scheduling policy for all threads currently running with *policy* to POLICY_TIMESHARE.

RETURN VALUE

KERN_SUCCESS
: The policy has been disabled.

KERN_INVALID_ARGUMENT
processor_set is not a valid processor set, or *policy* is not a recognized scheduling policy value, or an attempt was made to disable timesharing.

RELATED INFORMATION

Functions: **processor_set_policy_enable, thread_policy**.

processor_set_policy_enable

Function — Enables a scheduling policy for a processor set

LIBRARY

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

SYNOPSIS

```
kern_return_t processor_set_policy_enable
                (mach_port_t                    processor_set,
                 int                                   policy);
```

DESCRIPTION

The **processor_set_policy_enable** function extends the set of scheduling policies allowed for *processor_set*. The set of scheduling policies allowed for a processor set is the set of policies allowed to be set for threads assigned to that processor set. The current set of permitted policies can be obtained from **processor_set_info**.

PARAMETERS

processor_set
: [in scalar] The control port for the processor set for which a scheduling policy is to be enabled.

policy
: [in scalar] Policy to be enabled. The values currently defined are POLICY_TIMESHARE and POLICY_FIXEDPRI.

RETURN VALUE

KERN_SUCCESS
: The policy has been enabled.

KERN_INVALID_ARGUMENT
: *processor_set* is not a valid processor set, or *policy* is not a recognized scheduling policy value.

RELATED INFORMATION

Functions: **processor_set_policy_disable, thread_policy**.

processor_set_tasks

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

LIBRARY

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

SYNOPSIS

kern_return_t **processor_set_tasks**
(mach_port_t *processor_set,*
task_array_t* *task_list,*
mach_msg_type_number_t* *task_count*);

DESCRIPTION

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

PARAMETERS

processor_set
[in scalar] A processor set control port for which information is desired.

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

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

RETURN VALUE

KERN_SUCCESS
The information has been returned.

KERN_INVALID_ARGUMENT
processor_set is not a processor set port.

RELATED INFORMATION

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

processor_set_threads

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

LIBRARY

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

SYNOPSIS

kern_return_t **processor_set_threads**
(mach_port_t *processor_set,*
thread_array_t* *thread_list,*
mach_msg_type_number_t* *thread_count*);

DESCRIPTION

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

PARAMETERS

processor_set
[in scalar] A processor set control port for which information is desired.

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

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

RETURN VALUE

KERN_SUCCESS
The information has been returned.

KERN_INVALID_ARGUMENT
processor_set is not a processor set port.

RELATED INFORMATION

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

processor_start

Function — Start a processor

LIBRARY

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

SYNOPSIS

```
kern_return_t processor_start
                (mach_port_t                    processor);
```

DESCRIPTION

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

PARAMETERS

processor
: [in scalar] The processor to be controlled.

NOTES

This operation is machine dependent. It may do nothing.

CAUTIONS

The ability to restart an exited processor is machine dependent.

RETURN VALUE

KERN_SUCCESS
: The operation was performed.

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

KERN_INVALID_ARGUMENT
: *processor* is not a processor port.

RELATED INFORMATION

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

task_assign

Function — Assign a task to a processor set

LIBRARY

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

SYNOPSIS

```
kern_return_t task_assign
                (mach_port_t                    task,
                 mach_port_t                    processor_set,
                 boolean_t                      assign_threads);
```

DESCRIPTION

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

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

PARAMETERS

task
: [in scalar] The task to be assigned.

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

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

RETURN VALUE

KERN_SUCCESS
: The assignment was performed.

KERN_INVALID_ARGUMENT
: *task* is not a task port, or *processor_set* is not a processor set port for the same host as *task*.

RELATED INFORMATION

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

task_assign_default

Function — Assign a task to the default processor set

LIBRARY

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

SYNOPSIS

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

DESCRIPTION

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

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

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

PARAMETERS

task
: [in scalar] The task to be assigned.

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

RETURN VALUE

KERN_SUCCESS
: The assignment was performed.

KERN_INVALID_ARGUMENT
: *task* is not a task port.

RELATED INFORMATION

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

task_get_assignment

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

LIBRARY

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

SYNOPSIS

```
kern_return_t task_get_assignment
                (mach_port_t                    task,
                 mach_port_t*                   processor_set);
```

DESCRIPTION

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

PARAMETERS

task
: [in scalar] The task whose assignment is desired.

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

RETURN VALUE

KERN_SUCCESS
: The assigned set was returned.

KERN_INVALID_ARGUMENT
: *task* is not a task port.

KERN_INVALID_ADDRESS
: *processor_set* points to inaccessible memory.

RELATED INFORMATION

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

thread_assign

Function — Assign a thread to a processor set

LIBRARY

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

SYNOPSIS

kern_return_t **thread_assign**
(mach_port_t *thread,*
mach_port_t *processor_set*);

DESCRIPTION

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

PARAMETERS

thread
[in scalar] The thread to be assigned.

processor_set
[in scalar] The name port for the processor set into which the thread is to be assigned.

RETURN VALUE

KERN_SUCCESS
The assignment was performed.

KERN_INVALID_ARGUMENT
thread is not a thread port, or *processor_set* is not a processor set port for the same host as *thread.*

RELATED INFORMATION

Functions**: thread_assign_default, thread_get_assignment, processor_set_-create, processor_set_info, task_assign**, **processor_assign**.

thread_assign_default

Function — Assign a thread to the default processor set

LIBRARY

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

SYNOPSIS

kern_return_t **thread_assign_default**
(mach_port_t *thread*);

DESCRIPTION

The **thread_assign_default** function assigns *thread* to the default processor set. After the assignment is completed, the thread executes only on processors that are assigned to that processor set. Any previous assignment of the thread is nullified. This variant of **thread_assign** exists because the control port for the default processor set is privileged, and therefore not available to most tasks.

PARAMETERS

thread
: [in scalar] The thread to be assigned.

RETURN VALUE

KERN_SUCCESS
: The assignment was performed.

KERN_INVALID_ARGUMENT
: *thread* is not a thread port.

RELATED INFORMATION

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

thread_get_assignment

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

LIBRARY

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

thread
: [in scalar] The thread whose assignment is desired.

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

RETURN VALUE

KERN_SUCCESS
: The assigned set was returned.

KERN_INVALID_ARGUMENT
: *thread* is not a thread port.

KERN_INVALID_ADDRESS
: *processor_set* points to inaccessible memory.

RELATED INFORMATION

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

CHAPTER 10 Device Interface

This chapter discusses the specifics of the kernel's device interfaces. These interfaces provide read, write and status interfaces to devices.

device_close

Function — De-establish a connection to a device.

LIBRARY

#include <**device/device.h**>

SYNOPSIS

```
kern_return_t device_close
                (mach_port_t                    device);
```

DESCRIPTION

The **device_close** function decrements the open count for the named device. If this count reaches zero, the close operation of the device driver is invoked, closing the device.

PARAMETERS

device
: [in scalar] A device port to the device to be closed.

RETURN VALUE

D_SUCCESS
: Device was closed.

D_NO_SUCH_DEVICE
: *device* does not name a device port.

RELATED INFORMATION

Functions: **device_open**.

device_get_status

Function — Return the current device status

LIBRARY

#include <**device/device.h**>

SYNOPSIS

```
kern_return_t device_get_status
        (mach_port_t                        device,
         int                                flavor,
         dev_status_t                       status,
         mach_msg_type_number_t*            status_count);
```

DESCRIPTION

The **device_get_status** function returns status information pertaining to an open device. The possible values for *flavor* as well as the meaning of the returned status information is device dependent.

PARAMETERS

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

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

status
: [out array of *int*] The returned device status.

status_count
: [pointer to in/out scalar] On input, the reserved size of *status*; on output, the size of the returned device status.

RETURN VALUE

D_SUCCESS
: Status was returned.

D_NO_SUCH_DEVICE
: Device is not open or operational.

RELATED INFORMATION

Functions: **device_set_status**.

device_map

Function — Establish a memory manager representing a device

LIBRARY

#include **<device/device.h>**

SYNOPSIS

```
kern_return_t device_map
                (mach_port_t                    device,
                 vm_prot_t                      prot,
                 vm_offset_t                    offset,
                 vm_size_t                      size,
                 mach_port_t*                   pager,
                 int                            unmap);
```

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 pager port in a **vm_map** call. This call is device dependent.

PARAMETERS

device
: [in scalar] A device port to the device to be mapped.

prot
: [in scalar] Protection for the device memory.

offset
: [in scalar] An offset within the device memory object, in bytes.

size
: [in scalar] The size of the device memory object.

pager
: [out scalar] The returned abstract memory object port to a memory manager that represents the device.

unmap
: [in scalar] Currently unused.

NOTES

Port rights are maintained as follows:

Abstract memory object port:
: The device pager has all rights.

Memory cache control port:
: The device pager has only send rights.

Memory cache name port:
: The device pager has only send rights. The name port is not even recorded.

Regardless how the object is created, the control and name ports are created by the kernel and passed through the memory management interface.

CAUTIONS

The device pager assumes that access to its memory objects will not be propagated to more that one host, and therefore provides no consistency guarantees beyond those made by the kernel.

In the event that more than one host attempts to use a device memory object, the device pager will only record the last set of port names. [This can happen with only one host if a new mapping is being established while termination of all previous mappings is taking place.] Currently, the device pager assumes that its clients adhere to the initialization and termination protocols in the memory management interface; otherwise, port rights or out-of-line memory from erroneous messages may be allowed to accumulate.

RETURN VALUE

KERN_SUCCESS
: The device map is established.

D_NO_SUCH_DEVICE
: The device is not open or not operational.

RELATED INFORMATION

Functions: **vm_map**, **evc_wait**.

device_open

Function — Establish a connection to a device.

LIBRARY

#include **<device/device.h>** (**device_open**)

#include **<device/device_request.h>** (**device_open_request**)

#include **<device/device_reply.h>** (**ds_device_open_reply**)

SYNOPSIS

```
kern_return_t device_open
                (mach_port_t                    master_port,
                 dev_mode_t                     mode,
                 dev_name_t                     name,
                 mach_port_t*                   device);
```

DESCRIPTION

The **device_open** function opens a device object. The open operation of the device is invoked, if the device is not already open. The open count for the device is incremented.

ASYNCHRONOUS FORM

device_open_request

Function — Asynchronously request a connection to a device

```
kern_return_t device_open_request
                (mach_port_t                    master_port,
                 mach_port_t                    reply_port,
                 dev_mode_t                     mode,
                 dev_name_t                     name);
```

ds_device_open_reply

Server Interface — Receive the reply from an asynchronous open

```
kern_return_t ds_device_open_reply
                (mach_port_t                    reply_port,
                 kern_return_t                  return_code,
                 mach_port_t                    device);
```

PARAMETERS

master_port
: [in scalar] The master device port. This port is provided to the bootstrap task.

reply_port
: [in scalar] The port to which a reply is to be sent when the device is open.

mode
: [in scalar] Opening mode. This is the bit-wise OR of the following values:

  D_READ
  : Read access

  D_WRITE
  : Write access

  D_NODELAY
  : Do not delay on open

name
: [pointer to in array of *char*] Name of the device to open.

return_code
: [in scalar] Status of the open.

device
: [out scalar] The returned device port.

RETURN VALUE

D_SUCCESS
: Device was opened.

D_INVALID_OPERATION
: *master_port* is not the master device port.

D_WOULD_BLOCK
: The device is busy, but D_NOWAIT was specified in *mode*.

D_ALREADY_OPEN
: The device is already open in a mode incompatible with *mode*.

D_NO_SUCH_DEVICE
: *name* does not name a known device.

D_DEVICE_DOWN

The device has been shut down.

KERN_SUCCESS

Returned for **device_open_request** or **ds_device_open_reply**, since these functions do not receive a reply message and have no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **device_close**, **device_reply_server**.

device_read

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

LIBRARY

#include **<device/device.h>** (**device_read**)

#include **<device/device_request.h>** (**device_read_request**)

#include **<device/device_reply.h>** (**ds_device_read_reply**)

SYNOPSIS

```
kern_return_t device_read
        (mach_port_t                    device,
         dev_mode_t                     mode,
         recnum_t                       recnum,
         int                            bytes_wanted,
         io_buf_ptr_t*                  data,
         mach_msg_type_number_t*        data_count);
```

DESCRIPTION

The **device_read** function reads a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

ASYNCHRONOUS FORM

device_read_request

Function — 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,
         int                            bytes_wanted);
```

ds_device_read_reply

Server Interface — Receive the reply from an asynchronous read

```
kern_return_t ds_device_read_reply
        (mach_port_t                    reply_port,
         kern_return_t                  return_code,
         io_buf_ptr_t                   data,
         mach_msg_type_number_t         data_count);
```

PARAMETERS

device
[in scalar] A device port to the device to be read.

reply_port
[in scalar] The port to which the reply message is to be sent.

mode
[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT
Do not wait if data is unavailable.

recnum
[in scalar] Record number to be read.

bytes_wanted
[in scalar] Size of data transfer.

return_code
[in scalar] The return status code from the read.

data
[out pointer to dynamic array of bytes] Returned data bytes.

data_count
[out scalar] Number of returned data bytes.

RETURN VALUE

D_SUCCESS
Data was read.

D_NO_SUCH_DEVICE
The device is dead or not completely open.

KERN_SUCCESS
Returned for **device_read_request** or **ds_device_read_reply**, since these functions do not receive a reply message and have no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **device_read_inband**, **device_reply_server**.

device_read_inband

Function — Read a sequence of bytes “inband” from a device object.

LIBRARY

#include <**device/device.h**> (**device_read_inband**)

#include <**device/device_request.h**> (**device_read_request_inband**)

#include <**device/device_reply.h**> (**ds_device_read_reply_inband**)

SYNOPSIS

```
kern_return_t device_read_inband
        (mach_port_t                    device,
         dev_mode_t                     mode,
         recnum_t                       recnum,
         int                            bytes_wanted,
         io_buf_ptr_inband_t*           data,
         mach_msg_type_number_t*        data_count);
```

DESCRIPTION

The **device_read** function reads a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_read** in that the returned bytes are returned “inband” in the reply IPC message.

ASYNCHRONOUS FORM

device_read_request_inband

Function — Asynchronously read data

```
kern_return_t device_read_request_inband
        (mach_port_t                    device,
         mach_port_t                    reply_port,
         dev_mode_t                     mode,
         recnum_t                       recnum,
         int                            bytes_wanted);
```

ds_device_read_reply_inband

Server Interface — Receive the reply from an asynchronous read

```
kern_return_t ds_device_read_reply_inband
        (mach_port_t                    reply_port,
         kern_return_t                  return_code,
         io_buf_ptr_inband_t            data,
         mach_msg_type_number_t         data_count);
```

PARAMETERS

device
[in scalar] A device port to the device to be read.

reply_port
[in scalar] The port to which the reply message is to be sent.

mode
[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT
Do not wait if data is unavailable.

recnum
[in scalar] Record number to be read.

bytes_wanted
[in scalar] Size of data transfer.

return_code
[in scalar] The return status code from the read.

data
[out array of bytes] Returned data bytes.

data_count
[out scalar] Number of returned data bytes.

RETURN VALUE

D_SUCCESS
Data was read.

D_NO_SUCH_DEVICE
The device is dead or not completely open.

KERN_SUCCESS
Returned for **device_read_request_inband** or **ds_device_read_reply_inband**, since these functions do not receive a reply message and have no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **device_read, device_reply_server**.

device_set_filter

Function — Names an input filter for a device

LIBRARY

#include <**device/device.h**>

#include <**device/net_status.h**>

SYNOPSIS

kern_return_t **device_set_filter**

(mach_port_t	*device,*
mach_port_t	*receive_port,*
mach_msg_type_name_t	*receive_port_type,*
int	*priority,*
filter_array_t	*filter,*
mach_msg_type_number_t	*filter_count*);

DESCRIPTION

The **device_set_filter** function provides a means by which selected data appearing at a device interface can be selected and routed to a port.

The filter command list consists of an array of up to NET_MAX_FILTER (unsigned short) words to be applied to incoming messages to determine if those messages should be given to a particular input filter.

Each filter command list specifies a sequences of actions which leave a boolean value on the top of an internal stack. Each word 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 short word of the filter as the value.

NETF_PUSHZERO
: Use 0 as the value.

NETF_PUSHWORD+*N*
: Use short word *N* of the "data" portion of the message as the value.

NETF_PUSHHDR+*N*
: Use short word *N* of the "header" portion of the message as the value.

NETF_PUSHIND+*N*
: Pops the top long word from the stack and then uses short word *N* of the "data" portion of the message as the value.

NETF_PUSHHDRIND+*N*
: Pops the top long word from the stack and then uses short word *N* of the "header" portion of the message as the value.

NETF_PUSHSTK+*N*
: Use long word *N* of the stack (where the top of stack is long word 0) as the value.

NETF_NOPUSH
: Don't push a value.

The unsigned value so chosen is promoted to a long word before being pushed.

Once a value is pushed (except for the case of NETF_NOPUSH), the top two long words 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 scalar] A device port

receive_port
: [in scalar] The port to receive the input data that is selected by the filter.

receive_port_type
: [in scalar] IPC type of the send right provided to the device; either MACH_MSG_TYPE_MAKE_SEND or MACH_MSG_TYPE_MOVE_SEND.

priority
[in scalar] Used to order multiple receivers.

filter
[pointer to in array of *filter_t*] The address of an array of filter values.

filter_count
[in scalar] The size of the *filter* array.

RETURN VALUE

D_SUCCESS
Device filter set.

D_NO_SUCH_DEVICE
Device is not open or operational.

D_INVALID_OPERATION
No *receive_port* was supplied.

device_set_status

Function — Sets device status.

LIBRARY

#include <**device/device.h**>

SYNOPSIS

kern_return_t **device_set_status**

(mach_port_t	*device,*
int	*flavor,*
dev_status_t	*status,*
mach_msg_type_number_t	*status_count*);

DESCRIPTION

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

PARAMETERS

device
: [in scalar] 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 *int*] The status information to set.

status_count
: [in scalar] The size of the status information.

RETURN VALUE

D_SUCCESS
: Device status was set.

D_NO_SUCH_DEVICE
: The device is not open or operational.

RELATED INFORMATION

Functions: **device_get_status**.

device_write

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

LIBRARY

#include **<device/device.h>** (**device_write**)

#include **<device/device_request.h>** (**device_write_request**)

#include **<device/device_reply.h>** (**ds_device_write_reply**)

SYNOPSIS

```
kern_return_t device_write
                (mach_port_t                  device,
                 dev_mode_t                   mode,
                 recnum_t                     recnum,
                 io_buf_ptr_t                 data,
                 mach_msg_type_number_t       data_count,
                 int*                         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.

ASYNCHRONOUS FORM

device_write_request

Function — 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,
                 io_buf_ptr_t                 data,
                 mach_msg_type_number_t       data_count);
```

ds_device_write_reply

Server Interface — Receive the reply from an asynchronous write

```
kern_return_t ds_device_write_reply
                (mach_port_t                  reply_port,
                 kern_return_t                return_code,
                 int                          bytes_written);
```

PARAMETERS

device
[in scalar] A device port to the device to be written.

reply_port
[in scalar] The port to which the reply message is to be sent.

mode
[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT
Do not wait for I/O completion.

recnum
[in scalar] Record number to be written.

data
[pointer to in array of bytes] Data bytes to be written.

data_count
[in scalar] Number of data bytes to be written.

return_code
[in scalar] The return status code from the write.

bytes_written
[out scalar] Size of data transfer.

RETURN VALUE

D_SUCCESS
Data was written.

D_NO_SUCH_DEVICE
The device is dead or not completely open.

KERN_SUCCESS
Returned for **device_write_request** or **ds_device_write_reply**, since these functions do not receive a reply message and have no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **device_write_inband**, **device_reply_server**.

device_write_inband

Function — Write a sequence of bytes “inband” to a device object.

LIBRARY

#include **<device/device.h>** (**device_write_inband**)

#include **<device/device_request.h>** (**device_write_request_inband**)

#include **<device/device_reply.h>** (**ds_device_write_reply_inband**)

SYNOPSIS

kern_return_t **device_write_inband**
(mach_port_t *device,*
dev_mode_t *mode,*
recnum_t *recnum,*
io_buf_ptr_inband_t *data,*
mach_msg_type_number_t *data_count,*
int* *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. This call differs from **device_write** in that the bytes to be written are sent “inband” in the request IPC message.

ASYNCHRONOUS FORM

device_write_request_inband

Function — Asynchronously write data

kern_return_t **device_write_request_inband**
(mach_port_t *device,*
mach_port_t *reply_port,*
dev_mode_t *mode,*
recnum_t *recnum,*
io_buf_ptr_inband_t *data,*
mach_msg_type_number_t *data_count*);

ds_device_write_reply_inband

Server Interface — Receive the reply from an asynchronous write

kern_return_t **ds_device_write_reply_inband**
(mach_port_t *reply_port,*
kern_return_t *return_code,*
int *bytes_written*);

PARAMETERS

device
: [in scalar] A device port to the device to be written.

reply_port
: [in scalar] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait for I/O completion.

recnum
: [in scalar] Record number to be written.

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

data_count
: [in scalar] Number of data bytes to be written.

return_code
: [in scalar] The return status code from the write.

bytes_written
: [out scalar] Size of data transfer.

RETURN VALUE

D_SUCCESS
: Data was written.

D_NO_SUCH_DEVICE
: The device is dead or not completely open.

KERN_SUCCESS
: Returned for **device_write_request_inband** or **ds_device_write_reply_inband**, since these functions do not receive a reply message and have no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **device_write**, **device_reply_server**.

APPENDIX A MIG Server Routines

This appendix describes server message de-multiplexing routines generated by MIG from the kernel interface definitions of use to a server in handling messages sent from the kernel.

device_reply_server

Function — Handles messages from a kernel device driver

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
boolean_t device_reply_server
        (mach_msg_header_t*        in_msg,
         mach_msg_header_t*        out_msg);
```

DESCRIPTION

The **device_reply_server** function is the MIG generated server handling function to handle messages from kernel device drivers. Such messages were sent in response to the various **device_..._request**... calls. It is assumed when using those calls that some task is listening for reply messages on the port named as a reply port to those calls. The **device_reply_server** function performs all necessary argument handling for a kernel message and calls one of the device server functions to interpret the message.

PARAMETERS

in_msg
: [pointer to in structure] The device driver message received from the kernel.

out_msg
: [out structure] A reply message. No messages from a device driver expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this device handler interface and no other action was taken.

RELATED INFORMATION

Functions: **ds_device_open_reply**, **ds_device_write_reply**, **ds_device_write_reply_inband**, **ds_device_read_reply**, **ds_device_read_reply_inband**.

exc_server

Function — Handles kernel messages for an exception handler

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
boolean_t exc_server
        (mach_msg_header_t*                    in_msg,
         mach_msg_header_t*                    out_msg);
```

DESCRIPTION

The **exc_server** function is the MIG generated server handling function to handle messages from the kernel relating to the occurrence of an exception in a thread. Such messages are delivered to the exception port set via **thread_set_special_port** or **task_set_special_port**. When an exception occurs in a thread, the thread sends an exception message to its exception port, blocking in the kernel waiting for the receipt of a reply. The **exc_server** function performs all necessary argument handling for this kernel message and calls **catch_exception_raise**, which should handle the exception. If **catch_exception_raise** returns KERN_SUCCESS, a reply message will be sent, allowing the thread to continue from the point of the exception; otherwise, no reply message is sent and **catch_exception_raise** must have dealt with the exception thread directly.

PARAMETERS

in_msg
: [pointer to in structure] The exception message received from the kernel.

out_msg
: [out structure] A reply message.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the exception mechanism and no other action was taken.

RELATED INFORMATION

Functions: **thread_set_special_port**, **task_set_special_port**, **catch_exception_raise**.

memory_object_default_server

Function — Handles kernel messages for the default memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t memory_object_default_server
                (mach_msg_header_t*          in_msg,
                 mach_msg_header_t*          out_msg);
```

DESCRIPTION

The **memory_object_default_server** function is the MIG generated server handling function to handle messages from the kernel targeted to the default memory manager. This server function only handles messages unique to the default memory manager. Messages that are common to all memory managers are handled by **memory_object_server**.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **memory_object_default_server** function performs all necessary argument handling for a kernel message and calls one of the default memory manager functions.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **seqnos_memory_object_default_server**, **memory_object_server**, **memory_object_create**, **memory_object_data_initialize**.

memory_object_server

Function — Handles kernel messages for a memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t memory_object_server
        (mach_msg_header_t*          in_msg,
         mach_msg_header_t*          out_msg);
```

DESCRIPTION

The **memory_object_server** function is the MIG generated server handling function to handle messages from the kernel targeted to a memory manager.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **memory_object_server** function performs all necessary argument handling for a kernel message and calls one of the memory manager functions to interpret the message.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **memory_object_default_server**, **memory_object_copy**, **memory_-object_data_request**, **memory_object_data_unlock**, **memory_object_-data_write**, **memory_object_data_return**, **memory_object_init**, **memory_object_lock_completed**, **memory_object_change_completed**, **mem-ory_object_terminate**, **seqnos_memory_object_server**.

notify_server

Function — Handle kernel generated IPC notifications

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t notify_server
        (mach_msg_header_t*                    in_msg,
         mach_msg_header_t*                    out_msg);
```

DESCRIPTION

The **notify_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a **mach_msg** or **mach_port_request_notification** call. The **notify_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The notification message received from the kernel.

out_msg
: [out structure] Not used.

NOTES

The user of this function must also supply a dummy routine **do_mach_notify_port_deleted**, which will never be called, but which is defined as part of Mach 2.5 IPC compatibility.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE

The message did not apply to the notification mechanism and no other action was taken.

RELATED INFORMATION

Functions: **seqnos_notify_server**, **mach_msg**, **mach_port_request_notification,** **do_mach_notify_dead_name**, **do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_port_deleted**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once**.

seqnos_memory_object_default_server

Function — Handles kernel messages for the default memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t seqnos_memory_object_default_server
                (mach_msg_header_t*                    in_msg,
                 mach_msg_header_t*                    out_msg);
```

DESCRIPTION

The **seqnos_memory_object_default_server** function is the MIG generated server handling function to handle messages from the kernel targeted to the default memory manager. This server function only handles messages unique to the default memory manager. Messages that are common to all memory managers are handled by **seqnos_memory_object_server**.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **seqnos_memory_object_default_server** function performs all necessary argument handling for a kernel message and calls one of the default memory manager functions.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

NOTES

seqnos_memory_object_default_server differs from **memory_object_default_server** in that it supplies message sequence numbers to the server interfaces it calls.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **memory_object_default_server**, **seqnos_memory_object_server**, **seqnos_memory_object_create**, **seqnos_memory_object_data_initialize**.

seqnos_memory_object_server

Function — Handles kernel messages for a memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t seqnos_memory_object_server
        (mach_msg_header_t*                    in_msg,
         mach_msg_header_t*                    out_msg);
```

DESCRIPTION

The **seqnos_memory_object_server** function is the MIG generated server handling function to handle messages from the kernel targeted to a memory manager.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **seqnos_memory_object_server** function performs all necessary argument handling for a kernel message and calls one of the memory manager functions to interpret the message.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

NOTES

seqnos_memory_object_server differs from **memory_object_server** in that it supplies message sequence numbers to the server interfaces.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE

The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **seqnos_memory_object_default_server**, **seqnos_memory_object_-copy**, **seqnos_memory_object_data_request**, **seqnos_memory_object_data_unlock**, **seqnos_memory_object_data_write**, **seqnos_memory_object_data_return**, **seqnos_memory_object_init**, **seqnos_-memory_object_lock_completed**, **seqnos_seqnos_memory_object_change_completed**, **seqnos_memory_object_terminate**, **memory_object_server**.

seqnos_notify_server

Function — Handle kernel generated IPC notifications

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t seqnos_notify_server
        (mach_msg_header_t*        in_msg,
         mach_msg_header_t*        out_msg);
```

DESCRIPTION

The **seqnos_notify_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a **mach_msg** or **mach_port_request_notification** call. The **seqnos_notify_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The notification message received from the kernel.

out_msg
: [out structure] Not used.

NOTES

seqnos_notify_server differs from **notify_server** in that it supplies message sequence numbers to the server interfaces.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the notification mechanism and no other action was taken.

RELATED INFORMATION

Functions: **notify_server**, **mach_msg**, **mach_port_request_notification, do_seqnos_mach_notify_dead_name**, **do_seqnos_mach_notify_msg_accepted**, **do_seqnos_mach_notify_no_senders**, **do_seqnos_mach_notify_port_deleted**, **do_seqnos_mach_notify_port_destroyed**, **do_seqnos_mach_notify_send_once**.

APPENDIX B Multicomputer Support

Support for multicomputers is being added to the Mach kernel. This provides transparent support for distributed, non-shared-memory environments. The current support does not handle node failures and so is suitable to multicomputer environments but not yet to networked workstation environments.

With this support, a single logical Mach kernel is formed that spans a set of computers. The entire set acts as one Mach host. Each actual computer (possibly a multiprocessor) in the set, referred to as a *node*, is referenced by an integer node number within the containing "host".

This appendix describes operations that apply to individual nodes in such a configuration.

norma_get_special_port

Function — Returns a send right to a node specific port

LIBRARY

libmach_sa.a, **libmach.a**

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

SYNOPSIS

```
kern_return_t norma_get_special_port
        (mach_port_t                host_priv,
         int                        node,
         int                        which_port,
         mach_port_t*               special_port);
```

DESCRIPTION

The **norma_get_special_port** function returns a send right for a special port belonging to *node* on *host_priv*.

Each node maintains a (small) set of node specific ports. The device master port, host paging port, host name and host control port are maintained by the kernel. The kernel also permits a small set of server specified node specific ports; the name server port is an example and is given (by convention) an assigned special port index.

MACRO FORMS

norma_get_device_port

```
kern_return_t norma_get_device_port
        (mach_port_t                host_priv,
         int                        node,
         mach_port_t*               special_port)
```

⇒ **norma_get_special_port** (*host_priv*, *node*,
NORMA_DEVICE_PORT, *special_port*)

norma_get_host_paging_port

```
kern_return_t norma_get_host_paging_port
        (mach_port_t                host_priv,
         int                        node,
         mach_port_t*               special_port)
```

⇒ **norma_get_special_port** (*host_priv*, *node*,
NORMA_HOST_PAGING_PORT, *special_port*)

norma_get_host_port

kern_return_t **norma_get_host_port**

(mach_port_t	*host_priv,*
int	*node*,
mach_port_t*	*special_port)*

⇒ **norma_get_special_port** (*host_priv*, *node*,
NORMA_HOST_PORT, *special_port*)

norma_get_host_priv_port

kern_return_t **norma_get_host_priv_port**

(mach_port_t	*host_priv,*
int	*node*,
mach_port_t*	*special_port)*

⇒ **norma_get_special_port** (*host_priv*, *node*,
NORMA_HOST_PRIV_PORT, *special_port*)

norma_get_nameserver_port

kern_return_t **norma_get_nameserver_port**

(mach_port_t	*host_priv,*
int	*node*,
mach_port_t*	*special_port)*

⇒ **norma_get_special_port** (*host_priv*, *node*,
NORMA_NAMESERVER_PORT, *special_port*)

PARAMETERS

host_priv
[in scalar] The control port for the host for which to return the special port's send right.

node
[in scalar] The index of the node for which the port is desired.

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

NORMA_DEVICE_PORT
The device master port for the node.

NORMA_HOST_PAGING_PORT
The default pager port for the node.

NORMA_HOST_PORT
The host name port for the node. If the specified node is the current node, this value (unless otherwise set) is the same as would be returned by **mach_host_self**.

NORMA_HOST_PRIV_PORT
The host control port for the node.

NORMA_NAMESERVER_PORT
The registered name server port for the node.

special_port
[out scalar] The returned value for the port.

RETURN VALUE

KERN_SUCCESS
The port was returned.

KERN_INVALID_ARGUMENT
host_priv is not a valid host, *node* is not a valid node or *which_port* is not a valid port selector.

RELATED INFORMATION

Functions: **mach_host_self**, **norma_set_special_port**, **vm_set_default_memory_manager**.

norma_port_location_hint

Function — Guess a port's current location

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
kern_return_t norma_port_location_hint
                (mach_port_t                                    task,
                 mach_port_t                                    port,
                 int*                                           node);
```

DESCRIPTION

The **norma_port_location_hint** function returns the best guess of *port*'s current location. The hint is guaranteed to be a node where the port once was; it is guaranteed to be accurate if port has never moved. This can be used to determine residence node for hosts, tasks, threads, etc.

PARAMETERS

task
: [in scalar] Task reference (not currently used)

port
: [in scalar] Send right to the port to locate.

node
: [out scalar] Port location hint

RETURN VALUE

KERN_SUCCESS
: A hint was returned.

KERN_INVALID_ARGUMENT
: *port* is not a valid port.

RELATED INFORMATION

Functions: **task_set_child_node**, **norma_task_create**.

norma_set_special_port

Function — Sets a node specific special port

LIBRARY

libmach_sa.a, **libmach.a**

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

SYNOPSIS

kern_return_t **norma_set_special_port**
(mach_port_t *host_priv,*
int *node,*
int *which_port,*
mach_port_t *special_port)*;

DESCRIPTION

The **norma_set_special_port** function sets the special port belonging to *node* on *host_priv*.

Each node maintains a (small) set of node specific ports. The device master port, host paging port, host name and host control port are maintained by the kernel. The kernel also permits a small set of server specified node specific ports; the name server port is an example and is given (by convention) an assigned special port index.

MACRO FORMS

norma_set_device_port
kern_return_t **norma_set_device_port**
(mach_port_t *host_priv,*
int *node*,
mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*,
NORMA_DEVICE_PORT, *special_port*)

norma_set_host_paging_port
kern_return_t **norma_set_host_paging_port**
(mach_port_t *host_priv,*
int *node*,
mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*,
NORMA_HOST_PAGING_PORT, *special_port*)

norma_set_host_port

kern_return_t **norma_set_host_port**
(mach_port_t *host_priv,*
int *node*,
mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*,
NORMA_HOST_PORT, *special_port*)

norma_set_host_priv_port

kern_return_t **norma_set_host_priv_port**
(mach_port_t *host_priv,*
int *node*,
mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*,
NORMA_HOST_PRIV_PORT, *special_port*)

norma_set_nameserver_port

kern_return_t **norma_set_nameserver_port**
(mach_port_t *host_priv,*
int *node*,
mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*,
NORMA_NAMESERVER_PORT, *special_port*)

PARAMETERS

host_priv
[in scalar] The host for which to set the special port. Currently, this must be the per-node host control port.

node
[in scalar] The index of the node for which the port is to be set.

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

NORMA_DEVICE_PORT
The device master port for the node.

NORMA_HOST_PAGING_PORT
The default pager port for the node.

NORMA_HOST_PORT
The host name port for the node.

NORMA_HOST_PRIV_PORT
The host control port for the node.

NORMA_NAMESERVER_PORT
The registered name server port for the node.

special_port
[in scalar] A send right to the new special port.

RETURN VALUE

KERN_SUCCESS
The port was set.

KERN_INVALID_ARGUMENT
host_priv is not a valid host, *node* is not a valid node or *which_port* is not a valid port selector.

RELATED INFORMATION

Functions: **mach_host_self**, **norma_get_special_port**, **vm_set_default_memory_manager**.

norma_task_create

Function — Create a task on a specified node

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

kern_return_t **norma_task_create**

(mach_port_t	*parent_task,*
boolean_t	*inherit_memory*,
int	*child_node*,
mach_port_t*	*child_task)*;

DESCRIPTION

The **norma_task_create** function creates a new task from *parent_task* on the specified *node* and returns the name of the new task in *child_task*. The child task acquires shared or copied parts of the parent's address space (see **vm_inherit**). The child task initially contains no threads. The new task inherits the PC sampling status of its parent.

By way of comparison, tasks created by the standard **task_create** primitive are created on the node last set by **task_set_child_node** (by default the *parent_task*'s node).

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

- **task_kernel_port** — The port by which the kernel knows the new child task. The child task holds a send right for this port. The port name is also returned to the calling task.
- **task_bootstrap_port** — The port to which the child task can send a message requesting return of any system service ports that it needs (for example, a port to the Network Name Server or the Environment Manager). The child task inherits a send right for this port from the parent task. The child task can use **task_get_special_port** to change this port.
- **task_exception_port** — A default exception port for the child task, inherited from the parent task. The exception port is the port to which the kernel sends exception messages. Exceptions are synchronous interruptions to the normal flow of program control caused by the program itself. Some exceptions are handled transparently by the kernel, but others must be reported to the program. The child task, or any one of its threads, can change the default

exception port to take an active role in exception handling (see **task_get_special_port** or **thread_get_special_port**).

PARAMETERS

parent_task
: [in scalar] The task from which to draw the child task's port rights, resource limits, and address space.

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

child_node
: [in scalar] The node index of the node on which to create the child.

child_task
: [out scalar] The kernel-assigned name for the new task.

RETURN VALUE

KERN_SUCCESS
: A new task has been created.

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

KERN_RESOURCE_SHORTAGE
: Some critical kernel resource is unavailable.

RELATED INFORMATION

Functions: **task_set_child_node**, **task_create**.

task_set_child_node

Function — Set the node upon which future child tasks will be created

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

kern_return_t **task_set_child_node**
(mach_port_t *task,*
int *child_node)*;

DESCRIPTION

The **task_set_child_node** function specifies a node upon which child tasks will be created. This call exists only to allow testing with unmodified servers. Server developers should use **norma_task_create** instead.

PARAMETERS

task
[in scalar] The task who's children are to be affected.

node
[in scalar] The index of the node upon which future children should be created.

RETURN VALUE

KERN_SUCCESS
The node was set.

KERN_INVALID_ARGUMENT
task is not a valid task.

RELATED INFORMATION

Functions: **norma_task_create**.

APPENDIX C Intel 386 Support

This appendix describes special kernel interfaces to support the special hardware features of the Intel 386 processor and its successors.

Aside from the special functions listed here, the Intel 386 support also includes special thread state “flavors” (See **mach/thread_status.h**.).

- i386_THREAD_STATE—Basic machine thread state, except for segment and floating registers.
- i386_REGS_SEGS_STATE—Same as i386_THREAD_STATE but also sets/gets segment registers.
- i386_FLOAT_STATE—Floating point registers.
- i386_V86_ASSIST_STATE—Virtual 8086 interrupt table.

(The i386_ISA_PORT_MAP_STATE flavor shown in **mach/thread_status.h** has been disabled.)

IO Permission Bitmap

The 386 supports direct IO instructions. Generally speaking, these instructions are privileged (sensitive to IOPL). Mach, in combination with the processor, allows threads to directly execute these instructions against hardware IO ports for which the thread has permission (those named in its IO permission bitmap). (Note that this is a per-thread property.) The **i386_io_port_add** function enables IO to the port corresponding to the device port supplied to the call. **i386_io_port_remove** disables such IO; **i386_io_port_list** lists the devices to which IO is permitted.

For the sake of supporting the DOS emulator, the kernel supports a special device *iopl*. Access to this device implies access to the speaker, configuration CMOS, game port, sound blaster, printer and the VGA ports (device *kd0* or *vga*). Attempting to execute an IO instruction against one of these devices when the task holds send rights to the *iopl* device automatically adds these devices to the IO permission bitmap.

Virtual 8086 Support

Virtual 8086 mode is supported by Mach, enabled when the EFL_VM (virtual machine) flag in the thread state→*efl* is set. The various instructions sensitive to IOPL are simulated by the Mach kernel. This includes simulating an interrupt enabled flag and associated instructions.

A virtual 8086 task receives simulated 8086 interrupts by setting an interrupt descriptor table (in task space). This table is set with the i386_V86_ASSIST_STATE status flavor.

```
[1]  struct i386_v86_assist_state
[2]  {
[3]          unsigned int          int_table;
[4]          int                   int_count;
[5]  };
[6]  #define i386_V86_ASSIST_STATE_COUNT
             (sizeof (struct i386_v86_assist_state)/sizeof(unsigned int))
```

The *int_table* field points to an interrupt table in task space. The table has *int_count* entries. Each entry of this table has the format shown below.

```
[1]  struct v86_interrupt_table
[2]  {
[3]          unsigned int          count;
[4]          unsigned short        mask;
[5]          unsigned short        vec;
[6]  };
```

When the 8086 task has an associated interrupt table and its simulated interrupt enable flag is set, the kernel will scan the table looking for an entry whose *count* is greater than zero and whose *mask* value is not set. If found, the count will be decremented and the task will take a simulated 8086 interrupt to the address given by *vec*. No other simulated interrupts will be generated until the 8086 task executes an *iret* instruction and the (simulated) interrupt enable flag is again set. The generation of the simulated interrupt will turn off the hardware's trace trap flag; executing the *iret* instruction will restore the trace trap flag.

Local Descriptor Table

Although the 386 (and successors) view the address space as segmented, Mach provides each task with a linear address space (32 bits for the Intel family). The various entries in the system global descriptor table (GDT) are used for system use; in general the entries map all of kernel memory. The thread's local descriptor table (LDT) maps its task space. Segment 2 of this table is used for task code accesses (it permits only read access); segment 3 is used for data accesses (it permits write access, subject to page level protections); both segments, though, map all of the task's address space. Segment 1 of the table is unused. Segment 0 is used as a call gate for system calls (traps).

Each thread may set entries in its LDT to describe various ranges of its underlying address space. There is no way that this mechanism permits a thread to access any more virtual memory than its address space permits; these LDT segment entries merely provide different views of the address space. A segment may be thought of as an automatically relocated portion of the address space; the beginning of a segment can be referenced as address zero given the appropriately set 386 segment register. These local segment descriptors are manipulated with the **i386_set_ldt** function and examined with the **i386_get_ldt** function.

i386_get_ldt

Function — Return per-thread segment descriptors

LIBRARY

libmach_sa.a, **libmach.a**

#include <**mach/i386/mach_i386.h**>

SYNOPSIS

```
[1] struct descriptor
[2] {
[3]         unsigned int            low_word;
[4]         unsigned int            high_word;
[5] };
[6] typedef struct descriptor       descriptor_t;
[7] typedef struct descriptor*      descriptor_list_t;
```

```
kern_return_t i386_get_ldt
        (mach_port_t                        thread,
         int                                first_selector,
         int                                desired_count,
         descriptor_list_t*                 desc_list,
         mach_msg_type_number_t*            returned_count);
```

DESCRIPTION

The **i386_get_ldt** function returns per-thread segment descriptors from the thread's local descriptor table (LDT).

PARAMETERS

thread
: [in scalar] Thread whose segment descriptors are to be returned

first_selector
: [in scalar] Selector value (segment register value) corresponding to the first segment whose descriptor is to be returned

desired_count
: [in scalar] Number of returned descriptors desired

desc_list
: [unbounded out in-line array of *descriptor_t*] Array of segment descriptors. The reserved size of this array is supplied as the input value for *returned_count*.

returned_count
: [pointer to in/out scalar] On input, the reserved size of the descriptor array; on output, the number of descriptors returned

RETURN VALUE

KERN_SUCCESS
: Descriptors returned

KERN_INVALID_ARGUMENT
: Invalid *thread* or selector value out of range.

RELATED INFORMATION

Functions: **i386_set_ldt**.

i386_io_port_add

Function — Permit IO instructions to be performed against a device

LIBRARY

libmach_sa.a, **libmach.a**

#include <**mach/i386/mach_i386.h**>

SYNOPSIS

```
kern_return_t i386_io_port_add
                (mach_port_t                    thread,
                 mach_port_t                    device);
```

DESCRIPTION

The **i386_io_port_add** function adds a device to the IO permission bitmap for a thread, thereby permitting the thread to execute IO instructions against the device.

PARAMETERS

thread
: [in scalar] Thread whose permission bitmap is to be set.

device
: [in scalar] The device to which IO instructions are to be permitted.

NOTES

Normally, the thread must have called **i386_io_port_add** for all devices to which it will execute IO instructions. However, possessing send rights to the *iopl* device port will cause the *iopl* device to be automatically added to the thread's IO map upon first attempted access. This is a backward compatibility feature for the DOS emulator.

RETURN VALUE

KERN_SUCCESS
: The device was added to the IO permission bitmap.

KERN_INVALID_ARGUMENT
: *thread* or *device* were not valid.

RELATED INFORMATION

Functions: **i386_io_port_list**, **i386_io_port_remove**.

i386_io_port_list

Function — List devices permitting IO

LIBRARY

libmach_sa.a, **libmach.a**

#include <**mach/i386/mach_i386.h**>

SYNOPSIS

```
kern_return_t i386_io_port_list
                (mach_port_t                    thread,
                 device_list_t*                 list,
                 mach_msg_type_number_t*        count);
```

DESCRIPTION

The **i386_io_port_list** function returns a list of the devices named in the thread's IO permission bitmap, namely those permitting IO instructions to be executed against them.

PARAMETERS

thread
: [in scalar] Thread whose permission list is to be returned

list
: [out pointer to dynamic array of *device_t*] Device ports permitting IO

count
: [out scalar] Number of ports returned

RETURN VALUE

KERN_SUCCESS
: List returned

KERN_INVALID_ARGUMENT
: *thread* is invalid

KERN_RESOURCE_SHORTAGE
: Insufficient kernel memory to return list

RELATED INFORMATION

Functions: **i386_io_port_add**, **i386_io_port_remove**.

i386_io_port_remove

Function — Disable IO instructions against a device

LIBRARY

libmach_sa.a, **libmach.a**

#include <**mach/i386/mach_i386.h**>

SYNOPSIS

kern_return_t **i386_io_port_remove**
    (mach_port_t *thread,*
    mach_port_t *device)*;

DESCRIPTION

The **i386_io_port_remove** function removes the specified device from the thread's IO permission bitmap, thereby prohibiting IO instructions being executed against the device.

PARAMETERS

thread
: [in scalar] Thread whose permission bitmap is to be cleared

device
: [in scalar] Device whose permission is to be revoked

RETURN VALUE

KERN_SUCCESS
: Permission removed

KERN_INVALID_ARGUMENT
: *device* or *thread* was invalid

RELATED INFORMATION

Functions: **i386_io_port_add**, **i386_io_port_list**.

i386_set_ldt

Function — Set per-thread segment descriptors

LIBRARY

libmach_sa.a, **libmach.a**

#include <**mach/i386/mach_i386.h**>

SYNOPSIS

```
[1] struct descriptor
[2] {
[3]         unsigned int                low_word;
[4]         unsigned int                high_word;
[5] };
[6] typedef struct descriptor           descriptor_t;
[7] typedef struct descriptor*          descriptor_list_t;
```

```
kern_return_t i386_set_ldt
        (mach_port_t                    thread,
         int                            first_selector,
         descriptor_list_t              desc_list,
         mach_msg_type_number_t         count);
```

DESCRIPTION

The **i386_set_ldt** function allows a thread to have a private local descriptor table (LDT) which allows its local segments to map various ranges of its address space.

PARAMETERS

thread
: [in scalar] Thread whose segment descriptors are to be set

first_selector
: [in scalar] Selector value (segment register value) corresponding to the first segment whose descriptor is to be set

desc_list
: [pointer to in array of *descriptor_t*] Array of segment descriptors. The following forms are permitted:

- Empty descriptor. The ACC_P flag (segment present) may or may not be set.
- ACC_CALL_GATE — Converted into a system call gate. The ACC_P flag must be set.

All other descriptors must have both the ACC_P flag set and specify user mode access (ACC_PL_U).

- ACC_DATA
- ACC_DATA_W
- ACC_DATA_E
- ACC_DATA_EW
- ACC_CODE
- ACC_CODE_R
- ACC_CODE_C
- ACC_CODE_CR
- ACC_CALL_GATE_16
- ACC_CALL_GATE

count
[in scalar] Number of descriptors to be set

RETURN VALUE

KERN_SUCCESS
Descriptors set

KERN_INVALID_ARGUMENT
thread is invalid, the selector values are out of range or a segment descriptor is invalid

RELATED INFORMATION

Functions: **i386_get_ldt**.

APPENDIX D 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

```
[1]  struct host_basic_info
[2]  {
[3]          int                 max_cpus;
[4]          int                 avail_cpus;
[5]          vm_size_t           memory_size;
[6]          cpu_type_t          cpu_type;
[7]          cpu_subtype_t       cpu_subtype;
[8]  };
[9]  typedef struct host_basic_info      host_basic_info_data_t;
[10] 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

RELATED INFORMATION

Functions: **host_info.**

Data structures: **host_load_info**, **host_sched_info.**

host_load_info

Structure — Defines load information about a host

SYNOPSIS

```
[1]  #define CPU_STATE_USER              0
[2]  #define CPU_STATE_SYSTEM            1
[3]  #define CPU_STATE_IDLE              2
[4]  struct host_load_info
[5]  {
[6]          long                        avenrun[3];
[7]          long                        mach_factor[3];
[8]  };
[9]  typedef struct host_load_info       host_load_info_data_t;
[10] 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

mach_factor
: The processing resources available to a new thread—the number of CPUs divided by (1 + the number of threads)

RELATED INFORMATION

Functions: **host_info.**

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]		int	*min_timeout*;
[4]		int	*min_quantum*;
[5]	};		
[6]	typedef struct host_sched_info		**host_sched_info_data_t**;
[7]	typedef struct host_sched_info*		**host_sched_info_t**;

DESCRIPTION

The **host_sched_info** structure defines the scheduling information available about a host.

FIELDS

min_timeout
Minimum time-out, in milliseconds

min_quantum
Minimum quantum, in milliseconds

RELATED INFORMATION

Functions: **host_info.**

Data structures: **host_basic_info**, **host_load_info.**

mach_msg_header

Structure — Defines the header portion for messages

SYNOPSIS

```
[1]  typedef struct
[2]  {
[3]          mach_msg_bits_t          msgh_bits;
[4]          mach_msg_size_t          msgh_size;
[5]          mach_port_t              msgh_remote_port;
[6]          mach_port_t              msgh_local_port;
[7]          mach_port_seqno_t        msgh_seqno;
[8]          mach_msg_id_t            msgh_id;
[9]  } mach_msg_header_t;
```

DESCRIPTION

A Mach message consists of a fixed size message header, a **mach_msg_header_t**, followed by zero or more data items. Data items are typed. Each item has a type descriptor followed by the actual data (or an address of the data, for out-of-line memory regions).

There are two forms of type descriptors, a **mach_msg_type_t** and a **mach_msg_type_long_t**. The **mach_msg_type_long_t** type descriptor allows larger values for these fields. The *msgtl_header* field in the long descriptor is only used for its in-line, long-form, and de-allocate bits.

FIELDS

msgh_bits
: This field specifies the following properties of the message:

MACH_MSGH_BITS_REMOTE_MASK
: Encodes **mach_msg_type_name_t** values that specify the port rights in the *msgh_remote_port* field. The value must specify a send or send-once right for the destination of the message.

MACH_MSGH_BITS_LOCAL_MASK
: Encodes **mach_msg_type_name_t** values that specify the port rights in the *msgh_local_port* field. If the value doesn't specify a send or send-once right for the message's reply port, it must be zero and *msgh_local_port* must be MACH_PORT_NULL.

MACH_MSGH_BITS_COMPLEX

The complex bit must be specified if the message body contains port rights or out-of-line memory regions. If it is not specified, then the message body carries no port rights or memory, no matter what the type descriptors may seem to indicate.

MACH_MSGH_BITS_REMOTE(*bits*)

This macro returns the appropriate **mach_msg_type_name_t** values, given a *msgh_bits* value.

MACH_MSGH_BITS_LOCAL(*bits*)

This macro returns the appropriate **mach_msg_type_name_t** values, given a *msgh_bits* value.

MACH_MSGH_BITS (*remote*, *local*)

This macro constructs a value for *msgh_bits*, given two **mach_msg_type_name_t** values.

msgh_size

In the header of a received message, this field contains the message's size. The message size, a byte quantity, includes the message header, type descriptors, and in-line data. For out-of-line memory regions, the message size includes the size of the in-line address, not the size of the actual data region. There are no arbitrary limits on the size of a Mach message, the number of data items in a message, or the size of the data items.

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_seqno

The sequence number of this message relative to the port from which it is received. This field is ignored on sent messages.

msgh_id

Not set or read by the **mach_msg** call. The conventional meanings is to convey an operation or function id.

NOTES

Simple messages are provided to handle in-line data. The sender copies the in-line data into the message structure, and the receiver usually copies it out.

Non-simple messages are provided to handle out-of-line data. Out-of-line data allows for the sending of port information or data blocks that are very large or of variable size. The kernel maps out-of-line data from the address space of the sender to the address space of the receiver. The kernel copies the data only if the sender or receiver subsequently modifies it. This is an example of copy-on-write data sharing.

RELATED INFORMATION

Functions: **mach_msg**, **mach_msg_receive**, **mach_msg_send**.

Data Structures: **mach_msg_type**, **mach_msg_type_long**.

mach_msg_type

Structure — Defines the data descriptor for long data items in messages

SYNOPSIS

```
typedef struct
{
        unsigned int            msgt_name: 8,
                                msgt_size: 8,
                                msgt_number: 12,
                                msgt_inline: 1,
                                msgt_longform: 1,
                                msgt_deallocate: 1,
                                msgt_unused: 1;
} mach_msg_type_t;
```

DESCRIPTION

Each data item in a MACH IPC message has a type descriptor, a **mach_msg_type_t** or a **mach_msg_type_long_t**. The **mach_msg_type_long_t** type descriptor allows larger values for these fields.

FIELDS

msgt_name
: Specifies the data's type. The following types are predefined:

MACH_MSG_TYPE_UNSTRUCTURED
: un-interpreted data (32 bits)

MACH_MSG_TYPE_BIT
: single bit

MACH_MSG_TYPE_BOOLEAN
: boolean value (32 bits)

MACH_MSG_TYPE_INTEGER_16
: 16 bit integer

MACH_MSG_TYPE_INTEGER_32
: 32 bit integer

MACH_MSG_TYPE_CHAR
: single character

MACH_MSG_TYPE_BYTE
: 8-bit byte

MACH_MSG_TYPE_INTEGER_8
: 8-bit integer

MACH_MSG_TYPE_REAL
: floating value (32 bits)

MACH_MSG_TYPE_STRING
: null terminated

MACH_MSG_TYPE_STRING_C
: null terminated

MACH_MSG_TYPE_PORT_NAME
: type of **mach_port_t**. This is the type of the name for a port, not the type to specify if a port right is to be specified.

MACH_MSG_TYPE_MOVE_RECEIVE
: move the name receive right

MACH_MSG_TYPE_MOVE_SEND
: move the named send right

MACH_MSG_TYPE_MOVE_SEND_ONCE
: move the named send-once right

MACH_MSG_TYPE_COPY_SEND
: make a copy of the named send right

MACH_MSG_TYPE_MAKE_SEND
: make a send right from the named receive right

MACH_MSG_TYPE_MAKE_SEND_ONCE
: make a send-once right from the named send or receive right

The last six types specify port rights, and receive special treatment. The type MACH_MSG_TYPE_PORT_NAME describes port right names, when no rights are being transferred, but just names. For this purpose, it should be used in preference to MACH_MSG_TYPE_INTEGER_32.

msgt_size
: Specifies the size of each datum, in bits. For example, the *msgt_size* of MACH_MSG_TYPE_INTEGER_32 data is 32.

msgt_number
: Specifies how many data elements comprise the data item. Zero is a legitimate number. The total length specified by a type descriptor is (*msgt_size* * *msgt_number*), rounded up to an integral number of bytes. In-line data is then padded to an integral number of long-words. This en-

sures that type descriptors always start on long-word boundaries. It implies that message sizes are always an integral multiple of a long-word's size.

msgt_inline
When FALSE, specifies that the data actually resides in an out-of-line region. The address of the data region follows the type descriptor in the message body. The *msgt_name*, *msgt_size*, and *msgt_number* fields describe the data region, not the address.

msgt_longform
Specifies, when TRUE, that this type descriptor is a **mach_msg_type_long_t** instead of a **mach_msg_type_t.**

msgt_deallocate
Used with out-of-line regions. When TRUE, it specifies the data region should be de-allocated from the sender's address space (as if with **vm_deallocate**) when the message is sent.

msgt_unused
Not used, should be zero.

RELATED INFORMATION

Functions: **mach_msg**, **mach_msg_receive**, **mach_msg_send**.

Data Structures: **mach_msg_header**, **mach_msg_type_long**.

mach_msg_type_long

Structure — Defines the data descriptor for long data items in messages

SYNOPSIS

```
[1]  typedef struct
[2]  {
[3]          mach_msg_type_t          msgtl_header;
[4]          unsigned short           msgtl_name;
[5]          unsigned short           msgtl_size;
[6]          unsigned int             msgtl_number;
[7]  } mach_msg_type_long_t;
```

DESCRIPTION

Each data item has a type descriptor, a **mach_msg_type_t** or a **mach_msg_-type_long_t**. The **mach_msg_type_long_t** type descriptor allows larger values for these fields. The *msgtl_header* field in the long descriptor is only used for its in-line, long-form, and de-allocate bits.

FIELDS

msgtl_header

A header in common with **mach_msg_type_t**. When the *msgt_long-form* bit in the header is TRUE, this type descriptor is a **mach_msg_-type_long_t** instead of a **mach_msg_type_t**. The *msgt_name*, *msgt_size*, and *msgt_number* fields should be zero. Instead, **mach_msg** uses the following: *msgtl_name*, *msgtl_size*, and *msgtl_number* fields.

msgtl_name

Specifies the data's type. The defined values are the same as those for **mach_msg_type**.

msgtl_size

Specifies the size of each datum, in bits. For example, the *msgtl_size* of MACH_MSG_TYPE_INTEGER_32 data is 32.

msgtl_number

Specifies how many data elements comprise the data item. Zero is a legitimate number. The total length specified by a type descriptor is (*msgtl_size* * *msgtl_number*), rounded up to an integral number of bytes. In-line data is then padded to an integral number of long-words. This ensures that type descriptors always start on long-word boundaries. It implies that message sizes are always an integral multiple of a long-word's size.

RELATED INFORMATION

Functions: **mach_msg**, **mach_msg_receive**, **mach_msg_send**.

Data Structures: **mach_msg_header**, **mach_msg_type**.

mach_port_status

Structure — Defines information for a port

SYNOPSIS

```
struct mach_port_status
{
        mach_port_t             mps_pset;
        mach_port_seqno_t       mps_seqno;
        mach_port_mscount_t     mps_mscount;
        mach_port_msgcount_t    mps_qlimit;
        mach_port_msgcount_t    mps_msgcount;
        mach_port_rights_t      mps_sorights;
        boolean_t               mps_srights;
        boolean_t               mps_pdrequest;
        boolean_t               mps_nsrequest;
};
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

mps_seqno
Current sequence number for the port.

mps_mscount
Make-send count

mps_qlimit
Queue limit

mps_msgcount
Number in the queue

mps_sorights
How many send-once rights

mps_srights
True if send rights exist

mps_pdrequest
True if there is a port-deleted requested

mps_nsrequest
True if no-senders requested

RELATED INFORMATION

Functions: **mach_port_get_receive_status.**

mapped_time_value

Structure — Defines format of kernel maintained time in the mapped clock device

SYNOPSIS

```
struct mapped_time_value
{
        long                            seconds;
        long                            microseconds;
        long                            check_seconds;
};
typedef struct mapped_time_value        mapped_time_value_t;
```

DESCRIPTION

The **mapped_time_value** structure defines the format of the current-time structure maintained by the kernel and visible by mapping (**device_map**) the "time" pseudo-device. The data in this structure is updated at every clock interrupt. It contains the same value that would be returned by **host_get_time**.

FIELDS

seconds
: Seconds since system initialization

microseconds
: Microseconds in the current second

check_seconds
: A field used to synchronize with the kernel's setting of the time.

NOTES

Because of the race between the referencing of these multiple fields and the kernel's setting them, they should be referenced as follows:

```
do
{
        secs = mtime → seconds;
        usecs = mtime → microseconds;
} while (secs!= mtime → check_seconds);
```

RELATED INFORMATION

Functions: **device_map**, **host_adjust_time**, **host_get_time**, **host_set_time.**

processor_basic_info

Structure — Defines the basic information about a processor.

SYNOPSIS

```
[1]  struct processor_basic_info
[2]  {
[3]          cpu_type_t                   cpu_type;
[4]          cpu_subtype_t                cpu_subtype;
[5]          boolean_t                    running;
[6]          int                          slot_num;
[7]          boolean_t                    is_master;
[8]  };
[9]  typedef struct processor_basic_info*     processor_basic_info_t;
```

DESCRIPTION

The **processor_basic_info** structure defines the information available about a processor slot.

FIELDS

cpu_type
: Type of CPU

cpu_subtype
: Sub-type of CPU

running
: True if the CPU is running

slot_num
: Slot number of the CPU

is_master
: True if this is the master processor

RELATED INFORMATION

Functions: **processor_info**.

processor_set_basic_info

Structure — Defines the basic information about a processor set.

SYNOPSIS

```
[1] struct processor_set_basic_info
[2] {
[3]         int                              processor_count;
[4]         int                              task_count;
[5]         int                              thread_count;
[6]         int                              load_average;
[7]         int                              mach_factor;
[8] };
[9] 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

task_count
Number of tasks currently assigned to this processor set

thread_count
Number of threads currently assigned to this processor set

load_average
Scaled

mach_factor
Scaled

RELATED INFORMATION

Functions: **processor_set_info**.

Data Structures: **processor_set_sched_info**.

processor_set_sched_info

Structure — Defines the scheduling information about a processor set.

SYNOPSIS

```
[1]  struct processor_set_sched_info
[2]  {
[3]          int                                  policies;
[4]          int                                  max_priority;
[5]  };
[6]  typedef struct processor_set_sched_info*     processor_set_sched_info_t;
```

DESCRIPTION

The **processor_set_sched_info** structure defines the global scheduling information available about a processor set.

FIELDS

policies
: Allowed policies

max_priority
: Maximum scheduling priority for new threads

RELATED INFORMATION

Functions: **processor_set_info**.

Data Structures: **processor_set_basic_info**.

task_basic_info

Structure — Defines basic information for tasks

SYNOPSIS

```
[1]  struct task_basic_info
[2]  {
[3]          int                     suspend_count;
[4]          int                     base_priority;
[5]          vm_size_t               virtual_size;
[6]          vm_size_t               resident_size;
[7]          time_value_t            user_time;
[8]          time_value_t            system_time;
[9]  };
[10] 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.

base_priority
: The base scheduling priority for the task.

virtual_size
: The number of virtual pages for the task.

resident_size
: The number of resident pages for the task

user_time
: The total user run time for terminated threads within the task.

system_time
: The total system run time for terminated threads within the task.

RELATED INFORMATION

Functions: **task_info**.

Data Structures: **task_thread_times_info**.

task_thread_times_info

Structure — Defines thread execution times information for tasks

SYNOPSIS

```
[1] struct task_thread_times_info
[2] {
[3]         time_value_t                    user_time;
[4]         time_value_t                    system_time;
[5] };
[6] 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.

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
{
        time_value_t        user_time;
        time_value_t        system_time;
        int                 cpu_usage;
        int                 base_priority;
        int                 cur_priority;
        int                 run_state;
        int                 flags;
        int                 suspend_count;
        long                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.

system_time
The total system run time for the thread.

cpu_usage
Scaled CPU usage percentage for the thread.

base_priority
The base scheduling priority for the thread.

cur_priority
The current scheduling priority for the thread.

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: **thread_sched_info**.

thread_sched_info

Structure — Defines scheduling information for threads

SYNOPSIS

[1]	struct **thread_sched_info**	
[2]	{	
[3]	int	*policy*;
[4]	int	*data*;
[5]	int	*base_priority*;
[6]	int	*max_priority*;
[7]	int	*cur_priority*;
[8]	boolean_t	*depressed*;
[9]	int	*depress_priority*;
[10]	};	
[11]	typedef struct thread_sched_info*	**thread_sched_info_t**;

DESCRIPTION

The **thread_sched_info** structure defines the scheduling information array for threads. The **thread_info** function returns this array for a specified thread.

FIELDS

policy
Scheduling policy in effect

data
Associated data for the scheduling policy

base_priority
Base scheduling priority

max_priority
Maximum scheduling priority

cur_priority
Current scheduling priority

depressed
True if scheduling priority is depressed

depress_priority
Scheduling priority from which depressed

RELATED INFORMATION

Functions: **thread_info**.

Data Structures: **thread_basic_info**.

time_value

Structure — Defines format of system time values

SYNOPSIS

[1] struct **time_value**
[2] {
[3]   long   *seconds*;
[4]   long   *microseconds*;
[5] };
[6] typedef struct time_value   **time_value_t**;

DESCRIPTION

The **time_value** structure defines the format of the time structure supplied to or returned from the kernel.

FIELDS

seconds
Seconds since system initialization

microseconds
Microseconds in the current second

RELATED INFORMATION

Functions: **host_adjust_time**, **host_get_time**, **host_set_time.**

vm_statistics

Structure — Defines statistics for the kernel's use of virtual memory

SYNOPSIS

```
struct vm_statistics
{
        long                    pagesize;
        long                    free_count;
        long                    active_count;
        long                    inactive_count;
        long                    wire_count;
        long                    zero_fill_count;
        long                    reactivations;
        long                    pageins;
        long                    pageouts;
        long                    faults;
        long                    cow_faults;
        long                    lookups;
        long                    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.

You can also find *pagesize* by using the global variable *vm_page_size*. This variable is set at task initialization and remains constant for the life of the task.

For related information for a specific task, see the **task_basic_info** structure.

FIELDS

pagesize
: The virtual page size, in bytes.

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.

RELATED INFORMATION

Functions: **task_info**, **vm_statistics**.

Data Structures: **task_basic_info**.

APPENDIX E Error Return Values

This appendix lists the various kernel return values.

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* — Mach-IPC errors
- *err_local* — user defined errors

A typical user error code definition would be:

> #define SOMETHING_WRONG *err_local* **|** **err_sub** (13) **|** 1

D_ALREADY_OPEN

Exclusive-use device already open

D_DEVICE_DOWN

Device has been shut down

D_INVALID_OPERATION

Bad operation for device

D_INVALID_RECNUM

Invalid record (block) number

D_INVALID_SIZE

Invalid IO size

D_IO_ERROR

Hardware IO error

D_IO_QUEUED

IO queued - do not return result

D_NO_MEMORY

Memory allocation failure

D_NO_SUCH_DEVICE

No such device

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_SUCCESS

Normal device return

D_WOULD_BLOCK

Operation would block, but D_NOWAIT set

EML_BAD_CNT

Invalid syscall number

EML_BAD_TASK

Null task

KERN_ABORTED

The operation was aborted. IPC code will catch this and reflect it as a message error.

KERN_FAILURE

The function could not be performed; a catch-all.

KERN_INVALID_ADDRESS

Specified address is not currently valid.

KERN_INVALID_ARGUMENT

The function requested was not applicable to this type of argument, or an argument

KERN_INVALID_CAPABILITY

The supplied (port) capability is improper.

KERN_INVALID_HOST

Target host isn't actually a host.

KERN_INVALID_NAME

The name doesn't denote a right in the task.

KERN_INVALID_RIGHT

The name denotes a right, but not an appropriate right.

KERN_INVALID_TASK

Target task isn't an active task.

KERN_INVALID_VALUE

A blatant range error.

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_MEMORY_FAILURE

During a page fault, the target address refers to a memory object that has been destroyed. This failure is permanent.

KERN_NAME_EXISTS

The name already denotes a right in the task.

KERN_NO_ACCESS

Bogus access restriction.

KERN_NO_SPACE

The address range specified is already in use, or no address range of the size specified could be found.

KERN_NOT_IN_SET

The receive right is not a member of a port set.

KERN_NOT_RECEIVER

The task in question does not hold receive rights for the port argument.

KERN_PROTECTION_FAILURE

Specified memory is valid, but does not permit the required forms of access.

KERN_RESOURCE_SHORTAGE

A system resource could not be allocated to fulfill this request. This failure may not be permanent.

KERN_RIGHT_EXISTS

The task already has send or receive rights for the port under another name.

KERN_SUCCESS

Successful completion

KERN_UREFS_OVERFLOW

Operation would overflow limit on user-references.

MACH_MSG_IPC_KERNEL

(mask bit) Kernel resource shortage handling an IPC capability.

MACH_MSG_IPC_SPACE

(mask bit) No room in IPC name space for another capability name.

MACH_MSG_SUCCESS

Normal IPC success.

MACH_MSG_VM_KERNEL

(mask bit) Kernel resource shortage handling out-of-line memory.

MACH_MSG_VM_SPACE

(mask bit) No room in VM address space for out-of-line memory.

MACH_RCV_BODY_ERROR

Error receiving message body. See special bits.

MACH_RCV_HEADER_ERROR

Error receiving message header. See special bits.

MACH_RCV_IN_SET

Port is a member of a port set.

MACH_RCV_INTERRUPTED

Software interrupt.

MACH_RCV_INVALID_DATA

Bogus message buffer for in-line data.

MACH_RCV_INVALID_NAME

Bogus name for receive port/port-set.

MACH_RCV_INVALID_NOTIFY

Bogus notify port argument.

MACH_RCV_PORT_CHANGED

Port moved into a set during the receive.

MACH_RCV_PORT_DIED

Port/set was sent away/died during receive.

MACH_RCV_TIMED_OUT

Didn't get a message within the time-out value.

MACH_RCV_TOO_LARGE

Message buffer is not large enough for in-line data.

MACH_SEND_INTERRUPTED

Software interrupt.

MACH_SEND_INVALID_DATA

Bogus in-line data.

MACH_SEND_INVALID_DEST

Bogus destination port.

MACH_SEND_INVALID_HEADER

A field in the header had a bad value.

MACH_SEND_INVALID_MEMORY

Invalid out-of-line memory address.

MACH_SEND_INVALID_NOTIFY

Bogus notify port argument.

MACH_SEND_INVALID_REPLY

Bogus reply port.

MACH_SEND_INVALID_RIGHT

Bogus port rights in the message body.

MACH_SEND_INVALID_TYPE

Invalid msg-type specification.

MACH_SEND_MSG_TOO_SMALL

Data doesn’t contain a complete message.

MACH_SEND_NO_BUFFER

No message buffer is available.

MACH_SEND_NO_NOTIFY

Resource shortage; can’t request msg-accepted notification.

MACH_SEND_NOTIFY_IN_PROGRESS

Msg-accepted notification already pending.

MACH_SEND_TIMED_OUT

Message not sent before time-out expired.

MACH_SEND_WILL_NOTIFY

Msg-accepted notification will be generated.

MIG_ARRAY_TOO_LARGE

User specified array not large enough to hold returned array

MIG_BAD_ARGUMENTS

Server found wrong arguments

MIG_BAD_ID

Bad message ID

MIG_EXCEPTION

Server raised exception

MIG_NO_REPLY

Server shouldn't reply

MIG_REMOTE_ERROR

Server detected error

MIG_REPLY_MISMATCH

Wrong return message ID

MIG_SERVER_DIED

Server no longer exists

MIG_TYPE_ERROR

Type check failure

APPENDIX F

Index