OSF RI 1994 Mach Kernel Interfaces *Draft*

OSF Research Institute and
Carnegie Mellon University

August 18, 1994

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

Books in the OSF Mach series:

OSF Mach Kernel Principles

OSF Mach Kernel Interfaces

OSF Mach Server Writer's Guide

OSF Mach Server Library Interfaces

Revision History:

Revision 1.0 Draft: August 11, 1994

Revision 1.1 Draft: August 18, 1994

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

The format and basic contents of this document are derived from previous versions edited by Keith Loepere. This version was edited by Marc J. Teller.

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

Contents

CHAPTER 1 Introduction

This book documents the various interfaces to the 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 companion volumes.

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

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

Interface Descriptions

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

- The name of the interface
- A brief description

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

Interface Types

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

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

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_sa.a**) that has the calling sequence listed and that when called invokes some kernel or kernel related service. If the interface is a system trap instead of a message, it will be listed as a **System Trap**.

Some interfaces have the type **Server Interface**. Such a description applies to interfaces that are called in server tasks on behalf of messages sent from the kernel. That is, it is assumed that some task is listening (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 message field elements (such as sequence numbers). In most cases, a single server interface calling sequence has been chosen with a given MIG generated server message de-multiplexing routine that calls these interfaces. In some cases, there are more than one MIG generated server routines 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.
- The **Expanded** form defines an extended version of a function. When a function takes a large number of rarely used arguments, it may be expressed as an Expanded form of the commonly used interface. These Expanded forms are separate interfaces that take extra parameters at some loss in performance.
- The **Overwrite** form defines a version of a function that uses the IPC overwrite option to have the returned (out-of-line) data directly written over a user buffer.

Parameter Types

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

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

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

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

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

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

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

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

Error Return Values

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

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

Assuming the correct number and size of the parameters, the MIG stub will simply marshall these values, making no consistency checks. The stub then attempts to send this message using **mach_msg**. As such, the various IPC errors (MACH_SEND_...) are possible. In particular, if the destination port is completely bogus, the caller will receive MACH_SEND_INVALID_DEST. Note that most errors involving invalid rights or out-of-line memory addresses will be detected as IPC errors.

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

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

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

A return value of KERN_SUCCESS (or any other equivalent value) indicates that the requested operation was performed and any return values returned.

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.

Status:

Believed to be stable

MACH_SEND_CANCEL send option removed. **[OSF-RI DIPC]**

mach_msg

System Trap / Function — Sends and receives a message

SYNOPSIS

```
mach_msg_return_t mach_msg
        (mach_msg_header_t*         msg,
         mach_msg_option_t          option,
         mach_msg_size_t            send_size,
         mach_msg_size_t            receive_limit,
         mach_port_t                receive_name,
         mach_msg_timeout_t         timeout,
         mach_port_t                notify);
```

mach_msg_overwrite

Expanded form

```
mach_msg_return_t mach_msg_overwrite
        (mach_msg_header_t*         send_msg,
         mach_msg_option_t          option,
         mach_msg_size_t            send_size,
         mach_msg_size_t            receive_limit,
         mach_port_t                receive_name,
         mach_msg_timeout_t         timeout,
         mach_port_t                notify,
         mach_msg_header_t*         receive_msg,
         mach_msg_size_t            receive_msg_size);
```

DESCRIPTION

The **mach_msg** system call sends and receives Mach messages. Mach messages contain data, which can include port rights and addresses of large regions of memory. **mach_msg** uses the same buffer for sending and receiving a message; the other calls permit separate send and receive buffers (although they may be specified to be the same).

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

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

If the *option* argument contains both MACH_SEND_MSG and MACH_RCV_MSG, then **mach_msg** does both send and receive operations (in that order). If the send operation encounters an error (any return code other than

MACH_MSG_SUCCESS), then the call returns immediately without attempting the receive operation. Semantically the combined call is equivalent to separate send and receive calls, but it saves a system call and enables other internal optimizations.

If the *option* argument specifies neither MACH_SEND_MSG nor MACH_-RCV_MSG, then **mach_msg** does nothing.

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 containing random and reply rights] A message buffer used by **mach_msg** both for send and receive. This must be naturally aligned.

send_msg
: [pointer to in structure containing random and reply rights] The message buffer to be sent. This must be naturally aligned.

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

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

receive_limit
: [in scalar] When receiving a message, specifies the maximum size of the *msg* or *receive_msg* buffer in bytes. Otherwise zero should be supplied.

receive_name
: [in random right] 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 notify receive right] When using the MACH_SEND_CANCEL and MACH_RCV_NOTIFY options, specifies the port used for the notification. Otherwise MACH_PORT_NULL should be supplied.

receive_msg

[pointer to in/out structure] A message buffer into which a message (header and body) will be received. This must be naturally aligned. By default (**mach_msg**), any received message will overwrite the send message buffer. This buffer is in/out only if the MACH_RCV_OVERWRITE option is used; otherwise this buffer is out only.

receive_msg_size

[in scalar] When using the MACH_RCV_OVERWRITE option, specifies the size (in bytes) of the receive "message" that is to be used by **mach_msg** to indicate the disposition of received out-of-line regions.

NOTES

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

Major Concepts

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

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

The IPC implementation makes use of the VM system to efficiently transfer large amounts of data. The message can contain the addresses of regions of the sender's address space which should be transferred as part of the message. When a task receives a message containing such out-of-line regions of data, the data can appear in unused portions or overwrite an existing portion of the receiver's address space (depending on the requested receive options). Under favorable circumstances, 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 4 gigabytes 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 that the message contains a port right. If a message contains a receive right for a port, then the receive right is removed from the sender of the message and the right is transferred to the receiver of the message. While the receive right is in transit, tasks holding send rights can still send messages to the port, and they are queued until a task acquires the receive right and uses it to receive the messages.

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

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

Port Rights

Each task has its own space of port rights. Port rights are named with positive (unsigned) integers. For all architectures, sizeof (*mach_port_t*) = sizeof (*mach_port_name_t*) = sizeof (void*) and so user space addresses may be used as port names, except for the reserved values MACH_PORT_NULL (0) and MACH_PORT_DEAD (all 1 bits). When the kernel chooses a name for a new right, however, 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, or name 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.

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.

Tasks may hold multiple user-references for send rights. 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. Tasks may hold multiple user references for dead names.

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

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

MACH_MSG_TYPE_MAKE_SEND
: The message will carry a send right, but the caller must supply a receive right. The send right is created from the receive right, 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 must 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 must supply a send right. The user reference count for the supplied send right is decremented, and the right is destroyed if the count becomes zero. Unless a receive right remains, the name becomes available for recycling. 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 must 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 must supply a receive right. The caller loses the supplied receive right, but retains any send rights with the same name. The make-send count and sequence number of the receive right are reset to zero and no-more-senders notification requests are cancelled (with a send-once notification being sent to the no-more-senders notification right), but the port retains other attributes like queued messages and extant send and send-once rights.

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

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

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

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

It is also possible to send a (nearly unbounded) array of port rights "out-of-line". All of the rights named by the array must be of the same type. The array is physically copied with the message body proper. The array of port right (names) can be received by the receiver using the same options available for out-of-line data reception described below.

Memory

A message can contain one or more regions of the sender's address space which are to be transferred as part of the message. The message carries a logical copy of the memory. For this "out-of-line" memory, the kernel can copy the data or use virtual memory techniques to defer any actual page copies — unless the sender or the receiver modifies the data, the physical pages remain shared.

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

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

For each region, the sender has the choice of permitting the kernel to choose a transmission strategy or the choice of requiring physical copy:

MACH_MSG_VIRTUAL_COPY

In a sent message, this flag allows the kernel to choose any mechanism to transmit the data. For large regions, this involves constructing a virtual copy of the pages containing the region. The portion of the first page preceding the data and the portion of the last page following the data are not copied (and will appear as zero if the virtual copy is dynamically allocated in the receiver).

In a received message, this flag indicates that the kernel transmitted a virtual copy. Access to the received memory may involve interactions with the memory manager managing the sender's original data. Integrity-conscious receivers should exercise caution when dealing with out-of-line memory from un-trustworthy sources. Receivers concerned about deterministic access time should also exercise caution. The dynamic allocation option guarantees that the virtual copy will not be directly referenced during the act of receiving the message.

MACH_MSG_PHYSICAL_COPY

In a sent message, this flag requires that the kernel construct an actual copy of the memory (either into wired kernel memory or default memory managed space). There is a (fairly large) limit on the amount of data that can be physically copied in a message. Port arrays always assume this option when sent.

In a received message, this flag indicates that the kernel did transmit a physical copy.

The receiver has two options for the reception of out-of-line memory (or "out-of-line" port arrays): allocation and overwrite.

In the absence of the MACH_RCV_OVERWRITE option, all out-of-line regions are dynamically allocated. Allocated out-of-line memory arrives somewhere in the receiver's address space as new memory. It has the same inheritance and protection attributes as newly **vm_allocate**'ed memory. The receiver has the responsibility of de-allocating (with **vm_deallocate**) the memory when it is no longer needed. If the message contains more than one region, each will be allocated its own region, not necessarily contiguously. If the sender's data was transmitted as a virtual copy the allocated region will have the same data alignment within the page; otherwise, the received data will appear starting at the beginning of a page.

If the MACH_RCV_OVERWRITE option is set, the receiver can specify how each received region is to be processed (dynamically allocated as described above, or written over existing memory). With this option, the contents of the receive buffer (*receive_msg*) are examined by the kernel. The kernel scans the descriptors in the receive buffer "message" to determine how to handle each out-of-line region. (Note: whereas *receive_limit* is the maximum size of the receive buffer, *receive_msg_size* is the amount filled in with this "message".) The kernel uses each out-of-line data descriptor (in order) to specify the processing for each received data region in turn, each out-of-line port array descriptor is used correspondingly. (Intermingled port descriptors are ignored when matching descriptors between the incoming message and the receive buffer list.) The copy option in the matching descriptor specifies the processing:

MACH_MSG_OVERWRITE
: This flag indicates that the region should write over a specified region of the receiver's address space, as indicated by the address and size/count fields of the descriptor. The full range overwritten must already exist (be allocated or mapped) in the receiver's address space. Depending on the nature of the data transmission this overwrite may involve virtual memory manipulations or it may involve actual data copy.

MACH_MSG_ALLOCATE
: This flag indicates that the region is to be dynamically allocated. No other descriptor values are relevant.

If not enough descriptors appear in the receive buffer to describe all received regions, additional regions are dynamically allocated. If the receiver specifies more descriptors than there are regions in the received message, the additional descriptors are ignored (and do not appear in the final received message).

Note that the receive buffer descriptors will be overwritten:

- The size fields in descriptors will be updated (when scanned, they specified the maximum sizes of regions, when received, they specify the actual sizes of received regions).
- The copy fields in descriptors will be updated (when scanned, they specified allocate versus overwrite, when received, they indicate whether the region was physically or virtually copied).
- The descriptors may appear in different positions (given intermingled port descriptors).
- Descriptors that were not used (because there were not that many received regions) will be discarded.

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

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

Message Send

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

The message carries with it the security ID of the sender, which the receiver can request in the message trailer.

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

If the destination port's queue is full, 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 option is 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_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_TRAILER
: If set, the kernel, instead of determining the message attributes itself, will accept a formatted message trailer from the sender. The supplied trailer must be of the latest version supported by the kernel, and must contain all message attributes defined by the kernel. Only tasks with a security ID of KERNEL_SECURITY_ID can use this option; the intended use of this option is in support of the Net Message server. The trailer must follow the message in memory as it would appear in a received message. (The *send_size* argument to **mach_msg** still indicates the size of the message proper, not including this trailer.)

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.

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 option.

The intent of the pseudo-receive operation is to restore, as best as possible, the state prior to attempting the send. This involves restoring the port rights and out-of-line memory regions contained in the message. The port right names and out-of-line addresses in the message send buffer are updated to reflect the new values resulting from their effective reception. The pseudo-receive handles the destination and reply rights as any other rights; they are not reversed as is the appearance in a normal received message. Also, no trailer is appended to the 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.

Although unlikely, 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.

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 *receive_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 *receive_limit* argument specifies the size of the caller's message buffer (which must be big enough for the message header, body and trailer); the *msgh_size* field of the received message indicates the actual size of the received message header and body. The **mach_msg** call will not receive a message larger than *receive_limit*. Messages that are too large are destroyed, unless the MACH_RCV_LARGE option is used.

Following the received data, at the next natural boundary, is a message trailer. The *msgh_size* field of the received message does not include the length of this trailer; the trailer's length is given by the *msgh_trailer_size* field within the trailer. The receiver of a message is given a choice as to what trailer format is desired, and, within that format, which of the leading trailer attributes are desired (that is, to get trailer element three, the receiver must also accept elements one and two). For any given trailer format (of which there is currently only one), the trailer is compatibly extended by adding additional elements to the end.

Received messages are stamped (in the trailer) 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. (Note that this occurs whether or not the receiver requests the sequence number in the trailer.) 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.

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_LO-

CAL bits have a value of MACH_MSG_TYPE_PORT_SEND_ONCE or MACH_MSG_TYPE_PORT_SEND depending on the type of right to which the message was sent. The MACH_MSGH_BITS_REMOTE bits describe the reply port right.

A received message can contain port rights and out-of-line memory. The *msgh_local_port* field does not carry a port right; the act of receiving the message consumes the send or send-once right for the destination port. The *msgh_remote_port* field does carry a port right, and the message can carry additional port rights and memory if the MACH_MSGH_BITS_COMPLEX bit is set. 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 *receive_name*, or, if *receive_name* is a port set, a member of *receive_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_OVERWRITE
: If specified, the message buffer specified by *receive_msg* (or *msg*), of length *receive_msg_size*, will be scanned for out-of-line descriptors to specify the processing to be done when receiving out-of-line regions. This option is only allowed for **mach_msg_overwrite**.

MACH_RCV_LARGE

If the message is larger than *receive_limit* or an out-of-line region is larger than the size allowed by a corresponding receive descriptor (MACH_RCV_OVERWRITE), the message remains queued instead of being destroyed. If the header, trailer and body would not fit into *receive_limit*, only the message header (*mach_msg_header*) and trailer header (*mach_msg_trailer*) are returned with the actual size of the message returned in the *msgh_size* field, the actual size of the trailer returned in the *msgh_trailer_size* field and an error return value of MACH_RCV_TOO_LARGE. If *receive_limit* is sufficient but an out-of-line descriptor is not, the message header, trailer and body are received, with out-of-line descriptors set to indicate the nature and size of the out-of-line regions, with an error return of MACH_RCV_SCATTER_SMALL. No out-of-line regions or port rights (including the reply right) will be received. If this option is not specified, messages too large will be de-queued and then destroyed; the caller receives the message header, with all fields correct, including the destination port but excepting the reply port, which is MACH_PORT_NULL and an empty (no additional element) message trailer.

MACH_RCV_TRAILER_TYPE(*value*)

This macro encodes the type of trailer the kernel must return with the message. If the kernel does not recognize this type, it returns MACH_RCV_INVALID_TRAILER. Currently, only MACH_MSG_TRAILER_FORMAT_0 is supported.

MACH_RCV_TRAILER_ELEMENTS(*value*)

This macro encodes the number of trailer elements desired. If the kernel does not support this number for the requested trailer type, the kernel returns MACH_RCV_INVALID_TRAILER. Zero is a legal value. The following trailer elements are supported:

MACH_RCV_TRAILER_SEQNO

Returns the sequence number of the message relative to its port. This value is of type *mach_port_seqno_t*.

MACH_RCV_TRAILER_SENDER

Returns the security ID of the task that sent the message. This value is of type *security_id_t*.

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 corresponding element in the size array is set to zero. 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 header. The reply port and all port rights and memory in the message are destroyed. The caller receives the message 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 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 the message header atomically. Out-of-line memory and port rights in the message body do not enjoy this atomicity guarantee. These elements may be processed front-to-back, back-to-front, 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 port rights 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 rights 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 rights are 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. Either the reply port is dead on arrival or notification is requested.

Implementation

mach_msg and **mach_msg_overwrite** are wrappers for a system call. They have the responsibility for repeating the interrupted system call.

CAUTIONS

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

MACH_SEND_INVALID_REPLY
: The *msgh_local_port* value was invalid.

MACH_SEND_INVALID_TRAILER
: The trailer to be sent does not correspond to the current kernel format, or the sending task does not have the privilege to supply the message attributes.

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 kernel processed 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.

This return code implies that the message was queued:

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 *receive_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 *receive_name*.

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

MACH_RCV_TOO_LARGE
: When using MACH_RCV_LARGE, the message was larger than *receive_limit*. The message is left queued, and its actual size is returned in the message header/message body.

MACH_RCV_SCATTER_SMALL
: When using MACH_RCV_LARGE with MACH_RCV_OVERWRITE, one or more scatter list descriptors specified an overwrite region smaller than the corresponding incoming region. The message is left queued, and the proper descriptors are returned in the message header/message body.

MACH_RCV_INVALID_TRAILER
: The trailer type desired, or the number of trailer elements desired, is not supported by the kernel.

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_INVALID_DATA
: The specified message buffer was not writable.

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

MACH_RCV_SCATTER_SMALL
: When not using MACH_RCV_LARGE with MACH_RCV_OVERWRITE, one or more scatter list descriptors specified an overwrite region smaller than the corresponding incoming region. The message was de-queued and destroyed.

MACH_RCV_OVERWRITE_ERROR
: A region specified by a receive overwrite descriptor (MACH_RCV_OVERWRITE) was not allocated or could not be written.

MACH_RCV_INVALID_TYPE
: When using MACH_RCV_OVERWRITE, one or more scatter list descriptors did not have the type matching the corresponding incoming message descriptor or had an invalid copy (disposition) field.

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

These return codes imply that a message was received:

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

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 (bitwise-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: **vm_allocate**, **vm_deallocate**, **vm_write**, **mach_port_request_notification**.

Data Structures: **mach_msg_header**.

Status:

Experimental: [OSF-RI MK6]

mach_subsystem_create

Function — Register information about an RPC subsystem

SYNOPSIS

kern_return_t **mach_subsystem_create**

(mach_port_t	*target_task*,
user_subsystem_t	*user_subsys*,
mach_msg_type_number_t	*user_subsysCnt*,
mach_port_t *	*subsys*);

DESCRIPTION

The **mach_subsystem_create** function is used by a server to register information about an RPC subsystem with the kernel.

MIG automatically generates, in the server source file, a user_subsystem_t data structure that is appropriate for registering the subsystem. This data structure includes a per-routine array that specifies:

a) the address of the server function that performs the work.

b) the address of the MIG-generated server-side marshalling stub, to be used with **mach_msg**.

c) the argument signature (i.e. NDR-style argument format) of the routine.

Each port on which the server is to receive short-circuited RPC's (or a **mach_rpc** call) must be associated with a registered subsystem, by calling **mach_port_allocate_subsystem**.

A port naming the registered subsystem is returned.

PARAMETERS

target_task
: The task for which the subsystem is registered; normally the calling task, but not necessarily.

user_subsys
: The MIG-generated data structure describing the exported routines and their input/output characteristics (e.g. arguments and return values).

user_subsysCnt
: The size of the *user_subsys* data structure argument, in bytes.

subsys
: The port returned that names the registered subsystem.

RETURN VALUE

KERN_INVALD_ADDRESS
: One or more of the addresses in the range specified by the subsystem address and size are not valid addresses in the caller, or some of the internal pointers in the subsystem do not point to places within the address range (all of the data of the subsystem is required to be contiguous, even the parts that are pointed to by other parts).

KERN_INVALID_ARGUMENT
: The *target_task* port is not a send right naming a task, or the subsystem size is too small to be valid.

KERN_INVALID_TASK
: The target task is dead.

KERN_RESOURCE_SHORTAGE
: The kernel cannot allocate the subsystem due to insufficient available memory.

RELATED INFORMATION

Functions: **mach_rpc**, **mach_port_allocate_subsystem**, **thread_activation_-create**.

Structures: **user_subsystem**.

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.

Status:

Believed to be stable

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**
(mach_port_t *notify,*
mach_port_name_t *name)*;

do_seqnos_mach_notify_dead_name

Sequence Number form

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

DESCRIPTION

A **do_mach_notify_dead_name** function is called by **notify_server** as the result of a kernel message indicating that the port name is now dead as the result of the associated receive right having died. In contrast, a port-deleted notification indicates that the port name is no longer usable (that is, it no longer names a valid right), typically as a result of the right so named being consumed or moved. *notify* is the port named via **mach_port_request_notification** or **mach_msg**.

PARAMETERS

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

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

name
[in scalar] The dead name.

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **notify_server**, **seqnos_notify_server**, **mach_msg**, **mach_port_request_notification**, **do_mach_notify_no_senders**, **do_mach_notify_port_deleted**, **do_mach_notify_send_once.**

Status:

Believed to be stable except for the experimental proviso.

Proviso added about mach_port_allocate_full **[OSF-RI DIPC]**

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**

(mach_port_t	*notify,*
mach_port_mscount_t	*mscount,*
mach_port_name_t	*name)*;

do_seqnos_mach_notify_no_senders

Sequence Number form

kern_return_t **do_seqnos_mach_notify_no_senders**

(mach_port_t	*notify,*
mach_port_seqno_t	*seqno*,
mach_port_mscount_t	*mscount*,
mach_port_name_t	*name)*;

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**.

In order to receive a NO_MORE_SENDERS notification, the port generating it must have have been allocated using **mach_port_allocate_full**.

PARAMETERS

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

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

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

name
: [in scalar] The port name of the receive right at the time the notification was generated.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **notify_server**, **seqnos_notify_server**, **mach_msg**, **mach_port_request_notification**, **do_mach_notify_dead_name**, **do_mach_notify_port_deleted**, **do_mach_notify_send_once.**

Status:

Believed to be stable

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**
(mach_port_t *notify,*
mach_port_name_t *name)*;

do_seqnos_mach_notify_port_deleted

Sequence Number form

kern_return_t **do_seqnos_mach_notify_port_deleted**
(mach_port_t *notify,*
mach_port_seqno_t *seqno*,
mach_port_name_t *name)*;

DESCRIPTION

A **do_mach_notify_port_deleted** function is called by **notify_server** as the result of a kernel message indicating that a port name is no longer usable (that is, it no longer names a valid right), typically as a result of the right so named being consumed or moved. In contrast, a dead-name notification indicates that the port name is now dead as the result of the associated receive right having died. *notify* is the port named via **mach_port_request_notification** or **mach_msg**.

PARAMETERS

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

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

name
[in scalar] The invalid name.

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **notify_server**, **seqnos_notify_server**, **mach_msg**, **mach_port_request_notification, do_mach_notify_dead_name**, **do_mach_notify_no_senders**, **do_mach_notify_send_once.**

Status:

Believed to be stable

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**
(mach_port_t *notify)*;

do_seqnos_mach_notify_send_once

Sequence Number form

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

DESCRIPTION

A **do_mach_notify_send_once** function is called by **notify_server** as the result of a kernel message indicating that a send-once right was in any way destroyed. *notify* is the port for which a send-once right was destroyed.

PARAMETERS

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

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **notify_server**, **seqnos_notify_server**, **mach_msg**, **mach_port_request_notification**, **do_mach_notify_no_senders**, **do_mach_notify_port_deleted**, **do_mach_notify_dead_name.**

mach_port_allocate

Status:

Believed to be stable except for NMS.

Experimental: NO_MORE_SENDERS is not supported. **[OSF-RI DIPC]**

Function — Creates a port right

SYNOPSIS

kern_return_t **mach_port_allocate**
(task_port_t *task,*
mach_port_right_t *right,*
mach_port_name_t* *name);*

DESCRIPTION

The **mach_port_allocate** function creates a new right in the specified task. The new right's name is returned in *name*. The port allocated with this call provides a subset of Mach port semantics (i.e. NO_MORE_SENDERS detection is not provided). Requesting any feasture not supported will generate an error. Use **mach_port_allocate_full** to gain access to complete Mach port semantics.

PARAMETERS

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

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

MACH_PORT_RIGHT_RECEIVE
mach_port_allocate creates a port. The new port is not a member of any port set. It doesn't have any extant send or send-once rights. Its 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.

NOTES

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

RETURN VALUE

KERN_NO_SPACE

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

RELATED INFORMATION

Functions: **mach_port_allocate_full**, **mach_port_allocate_name**, **mach_port_deallocate, mach_port_insert_right, mach_port_extract_right**.

mach_port_allocate_full

Status:

Experimental. [OSF-RI DIPC]

Function — Creates a port right

SYNOPSIS

```
kern_return_t mach_port_allocate_full
                (task_port_t                    task,
                 mach_port_right_t              right,
                 mach_port_name_t*              name);
```

DESCRIPTION

The **mach_port_allocate_full** function creates a new right in the specified task. The new right's name is returned in *name*. The new port has all of the Mach port semantics (i.e. no_more_senders detection will work, if requested).

PARAMETERS

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

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

MACH_PORT_RIGHT_RECEIVE
: **mach_port_allocate_full** 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_full** creates a port set. The new port set has no members.

MACH_PORT_RIGHT_DEAD_NAME
: **mach_port_allocate_full** 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.

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_allocate_name**, **mach_port_deallocate**, **mach_port_insert_right**, **mach_port_extract_right**.

mach_port_allocate_name

Status:

Believed to be stable

Function — Creates a port right with a given name

SYNOPSIS

kern_return_t **mach_port_allocate_name**
(task_port_t *task,*
mach_port_right_t *right,*
mach_port_name_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 task send right] The task acquiring the port right.

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

MACH_PORT_RIGHT_RECEIVE
mach_port_allocate_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 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.

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_deallocate, mach_port_insert_right, mach_port_extract_right**.

mach_port_allocate_subsystem

Status:

Experimental: [OSF-RI MK6]

Function — Creates a port right and associates it with a subsystem

SYNOPSIS

```
kern_return_t mach_port_allocate_subsystem
                (task_port_t                 task,
                 mach_port_t                 subsys,
                 mach_port_t *               name);
```

DESCRIPTION

The **mach_port_allocate** function creates a new right and associates it with the specifed subsystem, *subsys*, previously registered with **mach_subsystem_create**. The new right's name is returned in *name*. The association of this port with the subsystem is immutable for the life of the port.

Any RPC targetd at the new port, *name*, will cause the kernel glue-code to locate the server function address and argument signature in the associated subsystem.

PARAMETERS

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

subsys
: [in scalar] The port right naming the subsystem the newly created port is to be associated with.

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

NOTES

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

RETURN VALUE

KERN_SUCCESS
: The right is allocated.

KERN_INVALID_TASK
: The space is null.

KERN_INVALID_TASK
The space is dead.

KERN_RESOURCE_SHORTAGE
Couldn't allocate memory.

KERN_NO_SPACE
No room in space for another right.

RELATED INFORMATION

Functions: **mach_subsystem_create**, **thread_activation_create**.

mach_port_deallocate

Status:

Believed to be stable

Function — Releases a user reference for a right

SYNOPSIS

```
kern_return_t mach_port_deallocate
                (task_port_t                          task,
                 mach_port_name_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. If *name* does not denote an element in the port name space, the function returns success.

PARAMETERS

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_RIGHT
: *name* denoted an invalid right.

RELATED INFORMATION

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

Status:

Believed to be stable

mach_port_destroy

Function — Removes a task's rights for a name

SYNOPSIS

kern_return_t **mach_port_destroy**
(task_port_t *task*;
mach_port_name_t *name)*;

DESCRIPTION

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

For most purposes, **mach_port_mod_refs** 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. 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 destruction of the send-once right produces a send-once notification for the port.

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

PARAMETERS

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

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

Status:

Believed to be stable

mach_port_extract_right

Function — Extracts a port right from a task

SYNOPSIS

kern_return_t **mach_port_extract_right**

(task_port_t	*task,*
mach_port_name_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 disposition for the right. 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 task send right] 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 random right] The extracted right.

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted an invalid right.

RELATED INFORMATION

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

Status:

Believed to be stable

Added note on true status always returned.
[OSF-RI DIPC]

mach_port_get_attributes

Function — Returns information about a port

SYNOPSIS

kern_return_t **mach_port_get_attributes**

(task_port_t	*task,*
mach_port_name_t	*name,*
mach_port_flavor_t	*flavor,*
mach_port_info_t	*port_info*,
mach_msg_type_number_t*	*port_info_count*);

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

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

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

NOTES

This interface is machine word length specific because of the port name parameter and the port set name in the MACH_PORT_RECEIVE_STATUS structure return.

Although lazy evaluation is permited for port death, **mach_port_get_attributes** will always the "true" status of a port with send or send-once rights.

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

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

RELATED INFORMATION

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

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

Status:

Believed to be stable

mach_port_get_refs

Function — Retrieves the number of user references for a right

SYNOPSIS

```
kern_return_t mach_port_get_refs
                (task_port_t                 task,
                 mach_port_name_t            name,
                 mach_port_right_t           right,
                 mach_port_urefs_t*          refs);
```

DESCRIPTION

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

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

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

RELATED INFORMATION

Functions: **mach_port_mod_refs**.

Status:

Believed to be stable

mach_port_get_set_status

Function — Returns the members of a port set

SYNOPSIS

kern_return_t **mach_port_get_set_status**

(task_port_t	*task,*
mach_port_name_t	*name,*
mach_port_name_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 task send right] 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_name_t*] The task's names for the port set's members.

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

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

RELATED INFORMATION

Functions: **mach_port_move_member**.

mach_port_insert_right

Status:

Believed to be stable

Function — Inserts a port right into a task

SYNOPSIS

```
kern_return_t mach_port_insert_right
                (task_port_t                    task,
                 mach_port_name_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 **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 task send right] The task which gets the caller's right.

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

right
: [in random right] 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.

NOTES

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

RETURN VALUE

KERN_NAME_EXISTS
: *name* already denoted a 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.

RELATED INFORMATION

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

Status:

Believed to be stable

mach_port_mod_refs

Function — Changes the number of user refs for a right

SYNOPSIS

kern_return_t **mach_port_mod_refs**

(task_port_t	*task,*
mach_port_name_t	*name,*
mach_port_right_t	*right,*
mach_port_delta_t	*delta)*;

DESCRIPTION

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

The *name* should denote the specified right. The number of user references for the right is changed by the amount *delta*, subject to the following restrictions: port sets, receive rights, 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 task send right] The task holding the right.

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

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

KERN_UREFS_UNDERFLOW
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**.

Status:

Believed to be stable

mach_port_move_member

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

SYNOPSIS

kern_return_t **mach_port_move_member**
(task_port_t *task,*
mach_port_name_t *member,*
mach_port_name_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 task send right] 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.

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
member or *after* did not denote a right.

KERN_INVALID_RIGHT
member denoted a right, but not a receive right, or *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_attributes**.

Status:

Believed to be stable

mach_port_names

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

SYNOPSIS

kern_return_t **mach_port_names**

(task_port_t	*task,*
mach_port_name_array_t*	*names,*
mach_msg_type_number_t*	*count*,
mach_port_type_array_t*	*types*);

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

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

count
: [out scalar] The number of names/types 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.

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_port_type**.

Status:

Believed to be stable

mach_port_rename

Function — Change a task's name for a right

SYNOPSIS

```
kern_return_t mach_port_rename
        (task_port_t                task,
         mach_port_name_t           old_name,
         mach_port_name_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 task send right] 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.

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *old_name* did not denote a right.

KERN_NAME_EXISTS
: *new_name* already denoted a right.

RELATED INFORMATION

Functions: **mach_port_names**.

mach_port_request_notification

Status:

Believed to be stable

Experimental:
Proviso added to NO_SENDERS semantics.
[OSF-RI DIPC]

Function — Request a notification of a port event

SYNOPSIS

```
kern_return_t mach_port_request_notification
                (task_port_t                 task,
                 mach_port_name_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_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 a send right is made directly from a receive right. The make-send count is reset to zero when the receive right is carried in a message. The notification also carries the receive right's name at the time the notification was generated.

When moving a receive right, no-senders notifications are canceled, with a send-once notification sent to indicate the cancelation.

The requested receive right must have been allocated with **mach_port_allocate_full**. Otherwise an error will be returned as it is an error to request NMS detection on a port that does not permit it (i.e. allocated with **mach_port_allocate**).

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted an invalid right. For example, requesting no_senders detection on a port allocated with **mach_port_allocate**.

KERN_INVALID_CAPABILITY
notify was invalid.

When using MACH_NOTIFY_DEAD_NAME:

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_msg**, **mach_port_get_attributes**.

Status:

Believed to be stable

mach_port_set_attributes

Function — Sets attributes of a port

SYNOPSIS

kern_return_t **mach_port_set_attributes**

(task_port_t	*task,*
mach_port_name_t	*name,*
mach_port_flavor_t	*flavor,*
mach_port_info_t	*port_info*,
mach_msg_type_number_t	*port_info_count*);

DESCRIPTION

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

PARAMETERS

task

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

name

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

flavor

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

MACH_PORT_LIMITS_INFO

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

port_info

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

port_info_count

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

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

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_allocate_name**, **mach_port_-get_attributes**.

Data Structures: **mach_port_limits**.

Status:

Believed to be stable

mach_port_set_mscount

Function — Changes the make-send count of a port

SYNOPSIS

```
kern_return_t mach_port_set_mscount
        (task_port_t                    task,
         mach_port_name_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*. *mscount* must be greater than or equal to zero.

PARAMETERS

task
: [in task send right] 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.

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

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

RELATED INFORMATION

Functions: **mach_port_get_attributes**.

mach_port_set_seqno

Status:

Believed to be stable

Function — Changes the sequence number of a port

SYNOPSIS

```
kern_return_t mach_port_set_seqno
                (task_port_t                    task,
                 mach_port_name_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 task send right] 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.

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

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

RELATED INFORMATION

Functions: **mach_port_get_attributes**

Status:

Believed to be stable

mach_port_type

Function — Return information about a task's port name

SYNOPSIS

```
kern_return_t mach_port_type
                (task_port_t                     task,
                 mach_port_name_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 send rights.

MACH_PORT_TYPE_RECEIVE
: The name denotes a receive right.

MACH_PORT_TYPE_SEND_ONCE
: The name denotes a send-once right.

MACH_PORT_TYPE_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.

PARAMETERS

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

name
: [in scalar] The name being queried.

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_names**, **mach_port_get_attributes**, **mach_port_get_-set_status**.

Status:

Believed to be stable

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.

If the task's task self port is null (thereby deactivating basic Mach manipulations by the task), this call returns null.

PARAMETERS

None

CAUTIONS

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

RETURN VALUE

MACH_PORT_NULL
: No port was allocated.

[reply receive right]
: Any other value indicates success.

RELATED INFORMATION

Functions: **mach_port_allocate**.

CHAPTER 4

Virtual Memory Interface

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

Status:

Believed to be stable

host_page_size

Function — Returns the current page size

SYNOPSIS

```
kern_return_t host_page_size
        (host_name_port_t        host,
         vm_offset_t*            page_size);
```

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_host_self**.

vm_allocate

Status:

Believed to be stable

Function — Allocates a region of virtual memory

SYNOPSIS

kern_return_t **vm_allocate**

(task_port_t	*target_task,*
vm_address_t*	*address,*
vm_size_t	*size,*
vm_allocate_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.

If *anywhere* is VM_ALLOCATE_ANYWHERE, the returned *address* will be at a page boundary; otherwise, the region starts at the beginning of the virtual page containing *address*. *size* is always rounded up to an integral number of pages. Because of this rounding to virtual page boundaries, the amount of memory allocated may be greater than *size*. Use **host_page_size** to find the current virtual page size.

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

PARAMETERS

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

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

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

anywhere
: [in scalar] Placement indicator. Valid values are:

VM_ALLOCATE_ANYWHERE
The kernel allocates the region in the next unused space that is sufficient within the address space. The kernel returns the starting address actually used in *address*.

VM_ALLOCATE_FIXED
The kernel allocates the region starting at *address* unless that space is reserved.

VM_ALLOCATE_RESERVED
The kernel allocates the region starting at *address* even if that space is reserved.

NOTES

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

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

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

RETURN VALUE

KERN_INVALID_ADDRESS
The specified address is illegal or reserved.

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

RELATED INFORMATION

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

vm_behavior_set

Status:

Believed to be stable

Function — Sets expected data access hints

SYNOPSIS

kern_return_t **vm_behavior_set**

(task_port_t	*target_task,*
vm_address_t	*address,*
vm_size_t	*size,*
vm_behavior_t	*behavior)*;

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

VM_BEHAVIOR_DEFAULT
: The system's default behavior. Assumes strong locality of reference, so LRU page replacement, possibly with pre-fetch, would be appropriate.

VM_BEHAVIOR_RANDOM
: No particular order expected. Assumes weak locality of reference, so LRU page replacement may be ineffective.

VM_BEHAVIOR_SEQUENTIAL
Forward sequential order.

VM_BEHAVIOR_RSEQNTL
Reverse sequential order.

NOTES

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

RETURN VALUE

KERN_INVALID_ADDRESS
The specified address is illegal or reserved.

RELATED INFORMATION

Functions: **vm_region, host_page_size**.

vm_copy

Status:

Believed to be stable

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

SYNOPSIS

```
kern_return_t vm_copy
        (task_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 the same task's virtual memory. It is semantically equivalent to **vm_read** followed by **vm_write**. The destination region can overlap the source region.

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

PARAMETERS

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

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

NOTES

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

RETURN VALUE

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**, **host_page_size**.

vm_deallocate

Status:

Believed to be stable

Function — Deallocates a region of virtual memory

SYNOPSIS

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

DESCRIPTION

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

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

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

PARAMETERS

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

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

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

NOTES

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

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

RETURN VALUE

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

RELATED INFORMATION

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

vm_inherit

Status:

Believed to be stable

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

SYNOPSIS

```
kern_return_t vm_inherit
                (task_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 **host_page_size** to find the current virtual page size.

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

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

Inheritance can be set for reserved regions. If the region is reserved but not allocated, the inheritance will be applied to any memory subsequently allocated or mapped in that region.

PARAMETERS

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

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

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

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

VM_INHERIT_SHARE
Allows child tasks to share the region.

VM_INHERIT_COPY
Gives child tasks a copy of the region.

VM_INHERIT_NONE
Provides no access to the region for child tasks.

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

vm_machine_attribute

Status:

Believed to be stable

Function — Sets and gets special attributes of a memory region

SYNOPSIS

```
kern_return_t vm_machine_attribute
                (task_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 underlying **pmap** module. These attributes are properties such as cachability, migratability and replicability. The behavior of this function is machine dependent.

PARAMETERS

target_task
: [in task send right] The port for 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. Aside from the generic values listed below, the following special values are defined:

MATTR_VAL_CACHE_FLUSH — flush from all caches

MATTR_VAL_DCACHE_FLUSH — flush from data caches

MATTR_VAL_ICACHE_FLUSH — flush from instruction caches

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. The new value can be a specific value listed above, or one of the following generic values:

MATTR_VAL_OFF
Turn attribute off.

MATTR_VAL_ON
Turn attribute on.

MATTR_VAL_GET
No change, just return current value.

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **vm_wire**.

vm_map

Status:

Believed to be stable

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

SYNOPSIS

```
kern_return_t vm_map
        (task_port_t              target_task,
         vm_address_t*            address,
         vm_size_t                size,
         vm_address_t             mask,
         vm_allocate_t            anywhere,
         mem_obj_rep_port_t       memory_object_representative,
         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 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.

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

PARAMETERS

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

address
: [pointer to in/out scalar] The starting address for the mapped object. The mapped object will start at the beginning of the page containing *address*. If there is not enough room following the address, the kernel does not map the object. The kernel returns the starting address actually used for the mapped object.

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

mask
: [in scalar] Alignment restrictions for starting address. Bits turned on in the mask will not be turned on in the starting address.

anywhere
: [in scalar] Placement indicator. Valid values are:

  VM_ALLOCATE_ANYWHERE
  : The kernel allocates the region in the next unused space that is sufficient within the address space. The kernel returns the starting address actually used in *address*.

  VM_ALLOCATE_FIXED
  : The kernel allocates the region starting at *address* unless that space is reserved.

  VM_ALLOCATE_RESERVED
  : The kernel allocates the region starting at *address* even if that space is reserved.

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

offset
[in scalar] An offset within the memory object, in bytes. The kernel maps *address* to the specified offset.

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

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

VM_PROT_READ
Allows read access.

VM_PROT_WRITE
Allows write access.

VM_PROT_EXECUTE
Allows execute access.

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

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

VM_INHERIT_SHARE
Allows child tasks to share the region.

VM_INHERIT_COPY
Gives child tasks a copy of the region.

VM_INHERIT_NONE
Provides no access to the region for child tasks.

NOTES

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

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

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

CAUTIONS

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

RETURN VALUE

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

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

KERN_INVALID_OBJECT
: The memory manager failed to map the memory object.

RELATED INFORMATION

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

vm_msync

Status:

Believed to be stable

Function — Synchronizes a memory range with its backing storage

SYNOPSIS

```
kern_return_t vm_msync
                (task_port_t                    target_task,
                 vm_address_t                   address,
                 vm_size_t                      size,
                 vm_sync_t                      sync_flags);
```

DESCRIPTION

The **vm_msync** function synchronizes the contents of a memory range with its backing store image by flushing or cleaning the contents of the specified range to the range's memory manager, engaging in a synchronization protocol with the manager (**memory_object_synchronize**). The client does not return from this call until the memory manager responds (to the kernel) with **memory_object_synchronize_completed**.

PARAMETERS

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

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

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

sync_flags
: [in scalar] The bit-wise OR of flags affecting the synchronization. Specifying both VM_SYNC_SYNCHRONOUS and VM_SYNC_ASYNCHRONOUS is invalid.

  VM_SYNC_INVALIDATE
  : Flushes pages in the range. Only precious pages are returned to the memory manager.

  VM_SYNC_SYNCHRONOUS
  : Writes dirty and precious pages back to the memory manager, waits for pages to reach backing storage.

VM_SYNC_ASYNCHRONOUS
: Writes dirty and precious pages back to the memory manager, returns without waiting for pages to reach backing storage.

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_synchronize**, **memory_object_synchronize_completed**.

vm_protect

Status:

Believed to be stable

Function — Sets access privileges for a region of virtual memory

SYNOPSIS

kern_return_t **vm_protect**

(task_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 **host_page_size** to find the current virtual page size.

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

Protection cannot be set for regions reserved but not allocated.

PARAMETERS

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

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

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

set_maximum
[in 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.

NOTES

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

RETURN VALUE

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

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

RELATED INFORMATION

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

vm_read

Status:

Believed to be stable

Function — Reads a task's virtual memory

SYNOPSIS

kern_return_t **vm_read**

(task_port_t	*target_task,*
vm_address_t	*address,*
vm_size_t	*size,*
pointer_t*	*data*,
mach_msg_type_number_t*	*data_count)*;

vm_read_overwrite

Overwrite form

kern_return_t **vm_read_overwrite**

(task_port_t	*target_task,*
vm_address_t	*address,*
vm_size_t	*size,*
pointer_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 task send right] The port for the task whose memory is to be read.

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

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

data
: [out pointer to dynamic array of bytes, pointer to in array of bytes for **vm_read_overwrite**] The array of data returned by the read.

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

NOTES

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

RETURN VALUE

KERN_PROTECTION_FAILURE
: Specified memory is valid, but does not permit 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

Status:

Believed to be stable

Function — Returns information on a region of virtual memory

SYNOPSIS

```
kern_return_t vm_region
        (task_port_t                 target_task,
         vm_address_t*               address,
         vm_size_t*                  size,
         vm_region_flavor_t          flavor,
         vm_region_info_t            info,
         mach_msg_type_number_t*     info_count);
```

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

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

info
: [out structure] Returned region information.

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

Data Structures: **vm_region_info**.

vm_remap

Status:

Believed to be stable

Function — Re-maps the memory objects in a range of a task's memory

SYNOPSIS

```
kern_return_t vm_remap
        (task_port_t          target_task,
         vm_address_t*        target_address,
         vm_size_t            size,
         vm_address_t         mask,
         vm_allocate_t        anywhere,
         task_port_t          source_task,
         vm_address_t         source_address,
         boolean_t            copy,
         vm_prot_t*           cur_protection,
         vm_prot_t*           max_protection,
         vm_inherit_t         inheritance);
```

DESCRIPTION

The **vm_remap** function maps the memory objects underlying a portion of the specified *source_task*'s virtual address space into the address space belonging to *target_task*. The target task can be the calling task or another task, identified by its task kernel port. The effect is as if the target task performed a **vm_map** call given the same memory object representative, maximum protection, current protection, inheritance and reservation state as the source task. However, the two tasks must reside on the same host. The kernel maps the memory objects starting at *target_address*, so that access to *target_address* is as if the source task accessed its *source_address*.

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 source task.
- 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* and *max_protection* parameters return the protection attributes for the mapped memory. If all memory within the range had the same attributes, these attributes are returned; otherwise the call returns the most restrictive values for any memory in the region.

PARAMETERS

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

target_address
: [pointer to in/out scalar] The starting address for the mapped memory in the target task. The mapped memory will start at the beginning of the page containing *target_address*. If there is not enough room following the address, the kernel does not map the memory. The kernel returns the starting address actually used for the mapped memory.

size
: [in scalar] The number of bytes to map. The kernel rounds this number up to an integral number of virtual pages.

mask
: [in scalar] Alignment restrictions for starting address. Bits turned on in the mask will not be turned on in the starting address.

anywhere
: [in scalar] Placement indicator. Valid values are:

VM_ALLOCATE_ANYWHERE
: The kernel allocates the region in the next unused space that is sufficient within the address space. The kernel returns the starting address actually used in *target_address*.

VM_ALLOCATE_FIXED
: The kernel allocates the region starting at *target_address* unless that space is reserved.

VM_ALLOCATE_RESERVED
: The kernel allocates the region starting at *target_address* even if that space is reserved.

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

source_address
: [in scalar] The starting address for the memory to be mapped from the source task. The memory to be mapped will start at the beginning of the page containing *source_address*. If not enough memory exists following the address, the kernel does not map the memory.

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

cur_protection
: [out scalar] The most restrictive current protection for the memory in 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
: [out scalar] The most restrictive maximum protection for the memory in 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

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

RETURN VALUE

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

KERN_PROTECTION_FAILURE
: Specified memory is valid, but the backing memory manager is not permitted by the requesting task.

RELATED INFORMATION

Functions: **vm_reserve**, **vm_map**, **vm_read**, **vm_write**.

vm_reserve

Status:

Believed to be stable

Function — Marks a memory region as reserved

SYNOPSIS

```
kern_return_t vm_reserve
                (task_port_t                    target_task,
                 vm_address_t                   address,
                 vm_size_t                      size);
```

DESCRIPTION

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

A reserved but not allocated region may have an inheritance value set but no other attributes. **vm_unreserve** will un-reserve the memory.

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_ADDRESS
: The address is illegal.

RELATED INFORMATION

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

Status:

Believed to be stable

vm_unreserve

Function — Marks a memory region as no longer reserved

SYNOPSIS

```
kern_return_t vm_unreserve
        (task_port_t          target_task,
         vm_address_t         address,
         vm_size_t            size);
```

DESCRIPTION

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

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_ADDRESS
: The address is illegal.

RELATED INFORMATION

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

vm_wire

Status:

Believed to be stable

Function — Specifies the pageability of a region of virtual memory

LIBRARY

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

SYNOPSIS

```
kern_return_t vm_wire
        (host_name_port_t        host_priv,
         task_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 the types of accesses to the memory region which must not suffer from (internal) faults of any kind after this call returns. A non-null *wired_access* value indicates that the page is to be "wired" into memory; a null value indicates "un-wiring". The kernel maintains for the region a count of the number of times the region is wired. A page is wired into physical memory if any task accessing it has a non-zero wired count for the page.

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

PARAMETERS

host_priv
: [in host-name send right] The name port for the host wiring the memory.

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

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

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

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.

NOTES

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

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **thread_wire**.

vm_write

Status:

Believed to be stable

Function — Writes data to a task's virtual memory

SYNOPSIS

kern_return_t **vm_write**

(task_port_t	*target_task,*
vm_address_t	*address,*
pointer_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.

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

PARAMETERS

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

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

data
: [pointer to page aligned in 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.

NOTES

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

RETURN VALUE

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

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.

Status:

Believed to be stable

memory_object_change_attributes

Function — Changes various object attributes

SYNOPSIS

kern_return_t **memory_object_change_attributes**

(mem_cache_control_port_t	*memory_control,*
memory_object_flavor_t	*flavor*,
memory_object_info_t	*attributes*,
mach_msg_type_number_t	*attributes_count*,
mach_port_t	*reply_to*);

DESCRIPTION

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

PARAMETERS

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

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

  MEMORY_OBJECT_ATTRIBUTES_INFO
  : Performance and behavior related attributes such as the cache indicator, cluster size and the copy strategy, sync invalidate flag. *attributes* should specify a structure of type **memory_object_attr_info**.

attributes
: [pointer to in structure] New attributes.

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

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

RETURN VALUE

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

RELATED INFORMATION

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

Data Structures: **memory_object_attr_info**.

Status:

Believed to be stable

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	*reply_port,*
mem_cache_control_port_t	*memory_control*,
memory_object_flavor_t	*flavor*);

seqnos_memory_object_change_completed

Sequence Number form

kern_return_t **seqnos_memory_object_change_completed**

(mach_port_t	*reply_port,*
mach_port_seqno_t	*seqno*,
mem_cache_control_port_t	*memory_control*,
memory_object_flavor_t	*flavor*);

DESCRIPTION

A **memory_object_change_completed** function is called as the result of a kernel message confirming the kernel's action in response to a **memory_object_change_attributes** call from the memory manager.

When the kernel completes the requested changes, it calls **memory_object_change_completed** (asynchronously) using the port explicitly provided in the **memory_object_change_attributes** call.

PARAMETERS

reply_port
: [in reply (receive) right] The port supplied in the corresponding **memory_object_change_attributes** call.

seqno
: [in scalar] The sequence number of this message relative to the port named in the **memory_object_change_attributes** call.

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

flavor

[in scalar] The flavor of attributes changed by the corresponding **memory_object_change_attributes** call.

NOTES

If the attribute change causes the termination of the object, no **memory_object_change_completed** message is sent. This case can be detected by the appearance of the **memory_object_terminate** message.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

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

Status:

Believed to be stable

memory_object_data_error

Function — Indicates no data for a memory object

SYNOPSIS

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

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

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.

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

Status:

Believed to be stable

memory_object_data_request

Server Interface — Requests data from a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_data_request
        (abs_mem_obj_port_t            memory_object,
         mem_cache_control_port_t      memory_control,
         vm_offset_t                   offset,
         vm_size_t                     length,
         vm_prot_t                     desired_access);
```

seqnos_memory_object_data_request

Sequence Number form

```
kern_return_t seqnos_memory_object_data_request
        (abs_mem_obj_port_t            memory_object,
         mach_port_seqno_t             seqno,
         mem_cache_control_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_notify** call.

The memory manager is expected to use **memory_object_data_supply** to return at least the specified data, with as much access as it can allow. If the memory manager cannot provide the data (for example, because of a hardware error), it can use the **memory_object_data_error** call. The memory manager can also use **memory_object_data_unavailable** to tell the kernel to supply zero-filled memory for the region.

PARAMETERS

memory_object
: [in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

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

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

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

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

desired_access
: [in scalar] The memory access modes to be allowed for the cached data. Possible values are obtained by or'ing together the following values:

  VM_PROT_READ
  : Allows read access.

  VM_PROT_WRITE
  : Allows write access.

  VM_PROT_EXECUTE
  : Allows execute access.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

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

Status:

Believed to be stable

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**

(abs_mem_obj_port_t	*memory_object,*
mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
pointer_t	*data,*
vm_size_t	*data_count*,
boolean_t	*dirty*,
boolean_t	*kernel_copy*);

seqnos_memory_object_data_return

Sequence Number form

kern_return_t **seqnos_memory_object_data_return**

(abs_mem_obj_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
pointer_t	*data,*
vm_size_t	*data_count*,
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.

PARAMETERS

memory_object

[in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

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

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

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

data
: [in pointer to dynamic array of bytes] The data that has been 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.

The kernel may re-request the returned data at any time following this message (including immediately) unless the “clean-in-place” mechanisms are in use for the memory object. In this case, the kernel will wait for a **memory_object_data_return_completed** message.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference and to de-allocate the returned data.

RELATED INFORMATION

Functions: **memory_object_data_supply**, **vm_deallocate**, **memory_object_synchronize**, **memory_object_server**, **seqnos_memory_object_server**, **memory_object_data_return_completed**.

Status:

Believed to be stable.

Will replace memory_-object_data_write_completed.
[OSF-RI NMK17]

memory_object_data_return_completed

Function — Inform the kernel that returned data has been disposed

SYNOPSIS

kern_return_t **memory_object_data_return_completed**
(mem_cache_control_port_t *memory_control,*
vm_offset_t *offset,*
vm_size_t *length*);

DESCRIPTION

The **memory_object_data_return_completed** function informs the kernel that previously returned data (**memory_object_data_return**) has been disposed. The memory manager need invoke this interface only if it explicitly requested the use of "clean-in-place" mechanisms when the memory object was initialized.

PARAMETERS

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

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

length
[in scalar] The amount of data disposed. The number must be an integral number of memory object pages.

CAUTIONS

The kernel's clean-in-place mechanism, available only to privileged memory managers, uses especially efficient means to return pages to the manager. The following conditions must be true for this mechanism to be in effect:

- The memory manager must be resource privileged and must assert this privilege by sending a **memory_object_establish_priv** message to the host control port instead of the host name port.
- The memory manager must be on the same host (node) as the memory cache object.
- The memory manager must not modify the returned pages.

The pages are returned in such a way that they will not be evicted from memory until the **memory_object_data_return_completed** message is received. As such, this mechanism must be used carefully to avoid exhausting the system's physical memory.

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_data_return**, **memory_object_data_request**, **memory_object_establish_priv**.

Status:

Believed to be stable

memory_object_data_supply

Function — Supplies data for a region of a memory object

SYNOPSIS

kern_return_t **memory_object_data_supply**

(mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
pointer_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 can only provide data that was requested by a **memory_object_data_request** call from the kernel.

PARAMETERS

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

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

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

NOTES

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

CAUTIONS

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

RETURN VALUE

Since this function 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_unavailable**, **memory_object_lock_request**, **memory_object_supply_completed**.

memory_object_data_unavailable

Status:

Believed to be stable

Function — Indicates no data for a memory object

SYNOPSIS

kern_return_t **memory_object_data_unavailable**
(mem_cache_control_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 is a normal user-created memory object. In this case, the kernel should provide zero-filled pages for the region.

PARAMETERS

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

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

RELATED INFORMATION

Functions: **memory_object_create**, **memory_object_data_error**, **memory_-object_data_request**, **memory_object_data_supply**.

memory_object_data_unlock

Status:

Believed to be stable

Server Interface — Requests access to a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_data_unlock**

(abs_mem_obj_port_t	*memory_object,*
mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_size_t	*length,*
vm_prot_t	*desired_access*);

seqnos_memory_object_data_unlock

Sequence Number form

kern_return_t **seqnos_memory_object_data_unlock**

(abs_mem_obj_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
mem_cache_control_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.

PARAMETERS

memory_object

[in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

seqno

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

memory_control

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

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

length
[in scalar] The number of bytes to which the access applies, starting at *offset*. The number converts to an integral number of memory object pages.

desired_access
[in scalar] The memory access modes requested for the cached data. Possible values are obtained by or'ing together the following values:

VM_PROT_READ
Allows read access.

VM_PROT_WRITE
Allows write access.

VM_PROT_EXECUTE
Allows execute access.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

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

memory_object_data_write_completed

Status:

Obsolete.

Will be replaced by memory_object_data_write_completed. **[OSF-RI NMK17]**

Function — Inform the kernel that returned data has been disposed

SYNOPSIS

kern_return_t **memory_object_data_write_completed**
(mem_cache_control_port_t *memory_control,*
vm_offset_t *offset,*
vm_size_t *length*);

DESCRIPTION

The **memory_object_data_write_completed** function informs the kernel that previously returned data (**memory_object_data_return**) has been disposed. The memory manager need invoke this interface only if it explicitly requested the use of "clean-in-place" mechanisms when the memory object was initialized.

PARAMETERS

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

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

length
: [in scalar] The amount of data disposed. The number must be an integral number of memory object pages.

CAUTIONS

The kernel's clean-in-place mechanism, available only to privileged memory managers, uses especially efficient means to return pages to the manager. The following conditions must be true for this mechanism to be in effect:

- The memory manager must be resource privileged and must assert this privilege by sending a **memory_object_establish_priv** message to the host control port instead of the host name port.
- The memory manager must be on the same host (node) as the memory cache object.
- The memory manager must not modify the returned pages.

The pages are returned in such a way that they will not be evicted from memory until the **memory_object_data_write_completed** message is received. As such, this mechanism must be used carefully to avoid exhausting the system's physical memory.

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_data_return**, **memory_object_data_request**, **memory_object_establish_priv**.

memory_object_destroy

Status:

Believed to be stable

Function — Shuts down a memory object

SYNOPSIS

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

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

Status:

Believed to be stable

mem_obj_rep and mem_cache_control arguments swapped **[OSF/1 R1.3].**

memory_object_establish

Function — Marks a memory object ready to receive paging operations

SYNOPSIS

kern_return_t **memory_object_establish**

(host_name_port_t	*host_name,*
mem_obj_rep_port_t	*memory_object_representative*,
mem_cache_control_port_t	*memory_control,*
abs_mem_obj_port_t	*abstract_memory_object,*
vm_prot_t	*representative_max_protection*,
security_id_t	*security_id*,
memory_object_flavor_t	*flavor*,
memory_object_info_t	*behavior*,
mach_msg_type_number_t	*behavior_count*);

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

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

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

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

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

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

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

NOTES

Destroying the abstract memory object port causes the memory object to be terminated. Destroying the memory object representative port does not terminate the memory object but it does terminate all mappings obtained through that representative.

RETURN VALUE

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

RELATED INFORMATION

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

Data Structures: **memory_object_behavior_info**.

Status:

Believed to be stable

mem_obj_rep and mem_cache_control arguments swapped **[OSF/1 R1.3].**

memory_object_establish_priv

Function — Marks a memory object ready to receive paging operations

SYNOPSIS

```
kern_return_t memory_object_establish_priv
        (host_control_port_t                          host_control,
        mem_obj_rep_port_t          memory_object_representative,
        mem_cache_control_port_t                    memory_control,
        abs_mem_obj_port_t                  abstract_memory_object,
        vm_prot_t                  representative_max_protection,
        security_id_t                                  security_id,
        memory_object_flavor_t                              flavor,
        memory_object_info_t                              behavior,
        mach_msg_type_number_t                      behavior_count);
```

DESCRIPTION

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

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

The use of this initialization response message (as opposed to **memory_object_establish**) indicates that the memory manager is resource privileged (with respect to kernel memory) and that the optimized "clean-in-place" mechanisms are to be used. The clean-in-place mechanisms permit significant kernel optimizations when returning data. However, the use of these mechanisms commits physical memory until the manager explicitly indicates that it has disposed of data returned to it (by sending a **memory_object_data_return_completed** message).

PARAMETERS

host_control
: [in host-control send right] The control port for the host upon which the object is being mapped. The manager determines this value from the host name port provided by the kernel in the **memory_object_notify** call.

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

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

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

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

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

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

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

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

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

NOTES

Destroying the abstract memory object port causes the memory object to be terminated. Destroying the memory object representative port does not terminate the memory object but it does terminate all mappings obtained through that representative.

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_change_attributes**, **memory_object_get_attributes**, **memory_object_notify**, **memory_object_representative**, **memory_object_rejected**, **memory_object_data_return_completed**, **memory_object_establish**, **memory_object_force_copy**.

Data Structures: **memory_object_behavior_info**.

memory_object_get_attributes

Status:

Believed to be stable

Function — Returns current attributes for a memory object

SYNOPSIS

kern_return_t **memory_object_get_attributes**

(mem_cache_control_port_t	*memory_control,*
memory_object_flavor_t	*flavor*,
memory_object_info_t	*attributes*,
mach_msg_type_number_t*	*attributes_count*);

DESCRIPTION

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

PARAMETERS

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

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

MEMORY_OBJECT_ATTRIBUTE_INFO
Performanceand behavior related attributes such as the cache indicator, the cluster size and the copy strategy, sync invalidate flag. *attributes* should specify a structure of type **memory_object_attr_info**.

attributes
[out structure] Current attributes.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **memory_object_change_attributes**, **memory_object_establish**.

Data Structures: **memory_object_attr_info**.

memory_object_lock_completed

Status:

Believed to be stable

Server Interface — Indicates completion of a consistency control call

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_lock_completed
                (mach_port_t                         reply_port,
                 mem_cache_control_port_t            memory_control,
                 vm_offset_t                         offset,
                 vm_size_t                           length);
```

seqnos_memory_object_lock_completed

Sequence Number form

```
kern_return_t seqnos_memory_object_lock_completed
                (mach_port_t                         reply_port,
                 mach_port_seqno_t                   seqno,
                 mem_cache_control_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.

PARAMETERS

reply_port
: [in reply (receive) right] The port supplied in the corresponding **memory_object_lock_request** call.

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

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

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

length
: [in scalar] The number of bytes to which the call refers, starting at *offset*. The number converts to an integral number of memory object pages.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

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

memory_object_lock_request

Status:

Believed to be stable

Function — Restricts access to memory object data

SYNOPSIS

kern_return_t **memory_object_lock_request**

(mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_size_t	*size,*
memory_object_return_t	*should_return,*
boolean_t	*should_flush,*
vm_prot_t	*lock_value,*
mach_port_t	*reply_port*);

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

offset

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

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_port

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

NOTES

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

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

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

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_force_copy**.

Status:

Believed to be stable

memory_object_notify

Server Interface — Announces an attempt by a kernel to establish a mapping via a memory object representative

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_notify
                (mem_obj_rep_port_t         memory_object_representative,
                 mem_cache_control_port_t   memory_control,
                 host_name_port_t           host_name,
                 vm_size_t                  memory_object_page_size);
```

seqnos_memory_object_notify

Sequence Number form

```
kern_return_t seqnos_memory_object_notify
                (mem_obj_rep_port_t         memory_object_representative,
                 mach_port_seqno_t          seqno,
                 mem_cache_control_port_t   memory_control,
                 host_name_port_t           host_name,
                 vm_size_t                  memory_object_page_size);
```

DESCRIPTION

A **memory_object_notify** function is called as the result of a kernel message notifying a memory manager that the kernel has been asked to map the specified memory object representative into a task's virtual address space.

When asked to map a memory object for the first time, the kernel responds by making a **memory_object_notify** call on the memory object representative. This call is the first message in a hand-shake dialog authenticating the memory manager (the receiver for the representative port) and the kernel to each other thereby establishing the identity and properties of the underlying memory object.

The call supplies the memory manager with the kernel generated memory cache control port which the memory manager must supply in its subsequent **memory_object_establish** call to authenticate itself to the kernel and which the memory manager will use to manipulate the memory cache object once initialization is complete. The supplied *host_name* port names the kernel mapping the object. The memory manager uses this port to verify the identity of the kernel sending the **memory_object_notify** message. This port is the destination of the subsequent **memory_object_establish** call.

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

The memory cache control port is initially created with default attributes. It (as well as the associated memory cache object) will be assigned its proper attributes as a result of the **memory_object_establish** call. No messages should be sent to the memory cache control port until memory object initialization is complete.

The call also supplies the virtual page size to be used for the memory mapping. The memory manager can use this size to detect mappings that use different data structures at initialization time, or to allocate buffers for use in reading data.

If a memory object is mapped into the address space of more than one task on different hosts (with independent kernels), the memory manager will receive a **memory_object_notify** call from each kernel, containing a unique set of host and object ports. Note that each kernel may also use a different page size.

PARAMETERS

memory_object_representative
: [in memory-object-representative (receive) right] The memory object representative port that represents the memory object data, as supplied to the kernel in a **vm_map** call.

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

memory_control
: [in memory-cache-control send right] A send right to the kernel generated memory cache control port to be used in the **memory_object_establish** reply.

host_name
: [in host-name send right] The name port for the host requesting the memory object initialization.

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 manage this object, it verifies that the supplied host name port does name a host and then responds to this port with a **memory_object_establish** call.

By virtue of the fact that the kernel sent this message, it did not know the association of the memory object representative to the underlying abstract memory object. If the abstract object underlying the memory representative is currently initialized, the manager should respond with **memory_object_representative** instead, since the supplied memory cache control port does not name the memory cache object for the abstract memory object.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the cache control and host name port references.

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_representative**, **memory_object_terminate**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_rejected

Status:

Believed to be stable

Server Interface — Indicates that a **memory_object_establish** message was discarded

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_rejected
                (abs_mem_obj_port_t                    memory_object,
                 mem_cache_control_port_t              memory_control,
                 kern_return_t                         reason);
```

seqnos_memory_object_rejected

Sequence Number form

```
kern_return_t seqnos_memory_object_rejected
                (abs_mem_obj_port_t                    memory_object,
                 mach_port_seqno_t                     seqno,
                 mem_cache_control_port_t              memory_control,
                 kern_return_t                         reason);
```

DESCRIPTION

A **memory_object_rejected** function is called as the result of a kernel message notifying a memory manager that a **memory_object_establish** message from the manager was ignored. This occurs when the manager believes it has finished initializing a memory object that either was not being initialized (a client faked a **memory_object_notify** message) or the kernel no longer wishes to initialize the object. The kernel makes this call to allow the memory manager to clean up data structures associated with the invalid mappings. In such cases, the *memory_control* port sent isn't really a memory cache control port. The kernel returns send rights to this port as a means of identifying which **memory_object_establish** message was discarded.

PARAMETERS

memory_object
: [in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

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

memory_control

[in random send right] A port that the memory manager thought was a memory cache control port. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

reason

[in scalar] The reason the memory object initialization failed.

KERN_INVALID_MEMORY_CONTROL

The memory cache control port supplied in the corresponding **memory_object_establish** or **memory_object_respresentative** message was not a memory cache control port.

KERN_INVALID_LEDGER

The ledger port supplied in the corresponding **memory_object_establish** or **memory_object_respresentative** message was not a ledger port.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the random port reference.

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_notify**, **memory_object_server**, **seqnos_memory_object_server**.

memory_object_representative

Status: **Believed to be stable**

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

SYNOPSIS

kern_return_t **memory_object_representative**

(host_name_port_t	*host_name,*
mem_obj_rep_port_t	*memory_object_representative*,
mem_cache_control_port_t	*memory_control,*
abs_mem_obj_port_t	*abstract_memory_object,*
vm_prot_t	*representative_max_protection*,
security_id_t	*security_id*);

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

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

NOTES

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

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

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

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

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

RETURN VALUE

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

RELATED INFORMATION

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

Status:

Believed to be stable

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                 reply_port,
         mem_cache_control_port_t    memory_control,
         vm_offset_t                 offset,
         vm_size_t                   length,
         kern_return_t               result,
         vm_offset_t                 error_offset);
```

seqnos_memory_object_supply_completed

Sequence Number form

```
kern_return_t seqnos_memory_object_supply_completed
        (mach_port_t                 reply_port,
         mach_port_seqno_t           seqno,
         mem_cache_control_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.

PARAMETERS

reply_port
: [in reply (receive) right] The port supplied in the corresponding **memory_object_data_supply** call.

seqno
: [in scalar] The sequence number of this message relative to the port named in the **memory_object_data_supply** call.

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

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

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

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

Status:

Believed to be stable

memory_object_synchronize

Server Interface — Indicates a client's request to update backing storage

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_synchronize**

(abs_mem_obj_port_t	*memory_object,*
mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_size_t	*length,*
vm_sync_t	*sync_flags*);

seqnos_memory_object_synchronize

Sequence Number form

kern_return_t **seqnos_memory_object_synchronize**

(abs_mem_obj_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
vm_size_t	*length,*
vm_sync_t	*sync_flags*);

DESCRIPTION

A **memory_object_synchronize** function is called as the result of a kernel message indicating that a client wishes to synchronize the contents of a range of a memory object with its backing storage image. This message would have been preceded by **memory_object_data_return** messages cleaning or flushing the specified range.

Depending on the client's supplied *sync_flags*, the manager waits for the pages to reach the desired state and then replies with **memory_object_synchronize_-completed** at which time the client returns from its **vm_msync** call. Multiple synchronize requests may be outstanding at a time but they will not overlap.

PARAMETERS

memory_object

[in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

seqno

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

memory_control

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

offset

[in scalar] The offset within the memory object.

length

[in scalar] The number of bytes cleaned or flushed, starting at *offset*. The number converts to an integral number of virtual pages.

sync_flags

[in scalar] The bit-wise OR of flags affecting the synchronization.

VM_SYNC_INVALIDATE

Flushes pages in the range. Only precious pages are returned to the memory manager.

VM_SYNC_SYNCHRONOUS

Writes dirty and precious pages back to the memory manager, waits for pages to reach backing storage.

VM_SYNC_ASYNCHRONOUS

Writes dirty and precious pages back to the memory manager, returns without waiting for pages to reach backing storage.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

Functions: **vm_msync**, **memory_object_synchronize_completed**, **memory_-object_data_return**, **memory_object_server**, **seqnos_memory_object_server**.

Status:

Believed to be stable

memory_object_synchronize_completed

Function — Inform the kernel that synchronized data has been processed

SYNOPSIS

kern_return_t **memory_object_synchronize_completed**
(mem_cache_control_port_t *memory_control,*
vm_offset_t *offset,*
vm_size_t *length*);

DESCRIPTION

The **memory_object_synchronize_completed** function informs the kernel that previously synchronized data (**memory_object_synchronize**) has been queued or placed on backing storage. This reply causes the issuing client to return from its **vm_msync** call. The offset and length must match that of the corresponding **memory_object_synchronize** call. There may be multiple synchronize requests for a given memory object outstanding but they will not overlap.

PARAMETERS

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

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

length
[in scalar] The amount of data processed. The number must be an integral number of memory object pages.

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_data_return**, **memory_object_synchronize**, **vm_msync**.

memory_object_terminate

Status:

Believed to be stable

Server Interface — Relinquishes access to a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_terminate**
(abs_mem_obj_port_t *memory_object,*
mem_cache_control_port_t *memory_control*);

seqnos_memory_object_terminate

Sequence Number form

kern_return_t **seqnos_memory_object_terminate**
(abs_mem_obj_port_t *memory_object,*
mach_port_seqno_t *seqno,*
mem_cache_control_port_t *memory_control*);

DESCRIPTION

A **memory_object_terminate** function is called as the result of a kernel message notifying a memory manager that no mappings of the specified memory object remain. The kernel makes this call to allow the memory manager to clean up data structures associated with the de-allocated mappings. The call provides receive rights to the memory cache control port so that the memory manager can retrieve any messages it sent into this port before knowing the memory object was being terminated and then destroy the port. The kernel also relinquishes its rights for all memory object ports.

The kernel terminates a memory object only after all address space mappings of the object have been de-allocated, or upon explicit request by the memory manager.

PARAMETERS

memory_object
[in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied to the kernel in a **memory_object_establish** call.

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

memory_control

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

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to remove the memory cache control port reference.

RELATED INFORMATION

Functions: **memory_object_destroy**, **memory_object_notify**, **mach_port_-deallocate**, **memory_object_server**, **seqnos_memory_object_server**.

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

Status:

Believed to be stable

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

LIBRARY

Not declared anywhere.

SYNOPSIS

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

catch_exception_raise_state

Expanded form

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

catch_exception_raise_state_identity

Expanded form

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

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NOTES

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

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

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

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

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

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

RETURN VALUE

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

RELATED INFORMATION

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

Status:

Believed to be stable

mach_thread_self

System Trap — Returns the thread self port

LIBRARY

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

SYNOPSIS

thread_port_t **mach_thread_self**
();

DESCRIPTION

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

PARAMETERS

None

RETURN VALUE

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

RELATED INFORMATION

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

receive_samples

Status:

Experimental support for the profil system call

Server Interface — Handles the occurrence of a PC sampling message

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **receive_samples**

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

DESCRIPTION

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

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_abort

Function — Aborts a thread

SYNOPSIS

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

DESCRIPTION

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

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

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

PARAMETERS

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

NOTES

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

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

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

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

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

CAUTIONS

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

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

RETURN VALUE

KERN_EXCEPTION_PROTECTED
: The thread is processing a protected exception.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_abort_safely

Function — Aborts a thread, restartably

SYNOPSIS

kern_return_t **thread_abort_safely**
(thread_port_t *target_thread)*;

DESCRIPTION

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

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

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

PARAMETERS

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

NOTES

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

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

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

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

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

CAUTIONS

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

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

RETURN VALUE

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

RELATED INFORMATION

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

Status:

Experimental: [OSF-RI MK6]

thread_activation_create

Function — Create a thread activation

SYNOPSIS

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

DESCRIPTION

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

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

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

PARAMETERS

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

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

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

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

NOTES

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_create

Function — Creates a thread within a task

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_create_running

Status:

Believed to be stable

Function — Optimized creation of a running thread

SYNOPSIS

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

DESCRIPTION

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

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

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

PARAMETERS

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

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

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

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_depress_abort

Function — Cancel thread scheduling depression

Status:

Believed to be stable

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_get_exception_ports

Function — Returns a send right to an exception port

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception.

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
: System call requested.

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

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

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

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

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

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

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

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

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

RETURN VALUE

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

RELATED INFORMATION

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

thread_get_special_port

Status:

Believed to be stable

Function — Returns a send right to a special port

SYNOPSIS

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

thread_get_kernel_port

Macro form

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

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

DESCRIPTION

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

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

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

PARAMETERS

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_get_state

Status:

Believed to be stable

Function — Returns the execution state for a thread

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_info

Function — Returns information about a thread

SYNOPSIS

kern_return_t **thread_info**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

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

thread_info
[out structure] Information about the specified thread.

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

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

Status:

Believed to be stable

thread_resume

Function — Resumes a thread

SYNOPSIS

```
kern_return_t thread_resume
                (thread_port_t                              target_thread);
```

DESCRIPTION

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

PARAMETERS

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_sample

Status:

Experimental support for the profil system call

Function — Perform periodic PC sampling for a thread

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_set_exception_ports

Function — Sets exception ports for a thread

SYNOPSIS

kern_return_t **thread_set_exception_ports**

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

DESCRIPTION

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

PARAMETERS

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

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

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception.

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
: System call requested.

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

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

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

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

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

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

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

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

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

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

NOTES

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_set_special_port

Status:

Believed to be stable

Function — Sets a special port for a thread

SYNOPSIS

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

thread_set_kernel_port

Macro form

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

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_set_state

Function — Sets the execution state for a thread

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_suspend

Status:

Believed to be stable

Function — Suspends a thread

SYNOPSIS

```
kern_return_t thread_suspend
        (thread_port_t                  target_thread);
```

DESCRIPTION

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

In this context, a user-level instruction can be either a machine instruction executed in user mode or a system trap instruction, including a page fault. If a thread is currently executing within a system trap, the kernel code may continue to execute until it reaches the system return code or it may suspend within the kernel code. In either case, the system trap returns when the thread resumes.

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

PARAMETERS

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

CAUTIONS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_swap_exception_ports

Function — Swaps exception ports for a thread

SYNOPSIS

kern_return_t **thread_swap_exception_ports**

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

DESCRIPTION

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

PARAMETERS

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

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

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
: System call requested.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NOTES

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

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

RETURN VALUE

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

RELATED INFORMATION

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

thread_switch

Status:

Believed to be stable

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

System Trap — Cause context switch with options

LIBRARY

Not declared anywhere.

SYNOPSIS

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

DESCRIPTION

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

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

SWITCH_OPTION_NONE
: The *time* argument is ignored.

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

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

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

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

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

PARAMETERS

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

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

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

NOTES

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

CAUTIONS

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

thread_terminate

Function — Destroys a thread

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_wire

Status:

Believed to be stable

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

LIBRARY

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

SYNOPSIS

kern_return_t **thread_wire**

(thread_port_t	*thread,*
host_control_port_t	*host_priv*,
boolean_t	*wired*);

DESCRIPTION

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

PARAMETERS

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

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

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **vm_wire**.

CHAPTER 7 Task Interface

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

Status:

Believed to be stable

mach_ports_lookup

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

SYNOPSIS

kern_return_t **mach_ports_lookup**

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

DESCRIPTION

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

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

PARAMETERS

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_ports_register**.

mach_ports_register

Status:

Believed to be stable

Function — Registers an array of well-known system ports

SYNOPSIS

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

DESCRIPTION

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

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

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

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

PARAMETERS

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

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

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

NOTES

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

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

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

mach_task_self

Status:

Believed to be stable

System Trap — Returns the task self port

LIBRARY

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

SYNOPSIS

```
task_port_t mach_task_self
                ();
```

DESCRIPTION

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

PARAMETERS

None

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **task_set_special_port**.

Status:

Believed to be stable

task_create

Function — Creates a task

SYNOPSIS

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

DESCRIPTION

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

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

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

The child task also inherits the following ports:

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

PARAMETERS

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

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

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

RETURN VALUE

Generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

task_create_security_id

Function — Creates a new task with an explicit security ID

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

Status:

Believed to be stable

task_get_emulation_vector

Function — Return user-level handlers for system calls.

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_set_emulation_vector**.

task_get_exception_ports

Status:

Believed to be stable

Function — Returns send rights to exception ports

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

  EXC_MASK_BAD_ACCESS
  : Could not access memory.

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

  EXC_MASK_ARITHMETIC
  : Arithmetic exception

  EXC_MASK_EMULATION
  : Emulation instruction. Emulation support instruction encountered.

  EXC_MASK_SOFTWARE
  : Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
: System call requested.

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

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

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

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

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

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

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

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

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

RETURN VALUE

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

RELATED INFORMATION

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

task_get_special_port

Status:

Believed to be stable

Function — Returns a send right to a special port

SYNOPSIS

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

task_get_bootstrap_port

Macro form

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

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

task_get_kernel_port

Macro form

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

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

task_get_host_name_port

Macro form

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

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

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

task_info

Function — Returns information about a task

SYNOPSIS

kern_return_t **task_info**

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

DESCRIPTION

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

PARAMETERS

task

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

flavor

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

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

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

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

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

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

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

task_info
[out structure] Information about the specified task.

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

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

Status:

Believed to be stable

task_resume

Function — Resume a task

SYNOPSIS

```
kern_return_t task_resume
        (task_port_t                    task);
```

DESCRIPTION

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

PARAMETERS

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

task_sample

Status:

Experimental support for the profil system call

Function — Perform periodic PC sampling for a task

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

task_set_emulation_vector

Function — Establishes user-level handlers for system calls.

SYNOPSIS

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

DESCRIPTION

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

These emulation handler addresses are inherited by child processes.

PARAMETERS

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

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

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

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_get_emulation_vector**.

Status:

Believed to be stable

task_set_exception_ports

Function — Sets exception ports for a task

SYNOPSIS

kern_return_t **task_set_exception_ports**

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

DESCRIPTION

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

PARAMETERS

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

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

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

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

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

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

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

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

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

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

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

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

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

NOTES

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

task_set_security_id

Status:

Believed to be stable

Function — Changes a task's security ID

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

security_id
: [in scalar] The new security ID.

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

Status:

Believed to be stable

task_set_special_port

Function — Sets a special port for a task

SYNOPSIS

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

task_set_bootstrap_port

Macro form

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

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

task_set_kernel_port

Macro form

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

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

task_set_host_name_port

Macro form

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

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Status:

Believed to be stable

task_suspend

Function — Suspends a task

SYNOPSIS

```
kern_return_t task_suspend
                (task_port_t                    task);
```

DESCRIPTION

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

PARAMETERS

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

task_swap_exception_ports

Status:

Believed to be stable

Function — Swaps exception ports for a task

SYNOPSIS

kern_return_t **task_swap_exception_ports**

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

DESCRIPTION

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

PARAMETERS

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

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

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

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

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

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

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

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

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

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

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

EXCEPTION_STATE_IDENTITY_PROTECTED
Send a **catch_exception_raise_state_identity** message including the thread identity and state. Mark the exception port (and associated exceptions) as protected.

flavor
[in scalar] The type of state to be sent with the exception message. These types are defined in **<mach/thread_states.h>**.

old_exception_masks
: [out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

old_exception_count
: [pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_exception_ports
: [out array of exception send rights] The returned exception ports.

old_behaviors
: [out array of *exception_behavior_t*] The type of exception message to be sent as with *behavior.*

old_flavors
: [out array of *thread_state_flavor_t*] The type of state to be sent with the exception message. These types are defined in **<mach/thread_-states.h>**.

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected; any other value causes the attempted execution of these system call numbers to be considered an exception.

A "protected" exception port is one which cannot be fetched and for which exception processing cannot be aborted (**thread_abort**).

RETURN VALUE

KERN_EXCEPTION_PROTECTED
: One of the requested exception ports is protected and cannot be returned.

RELATED INFORMATION

Functions: **mach_task_self**, **task_get_exception_ports**, **task_set_exception_ports**, **thread_create**, **thread_get_exception_ports**, **thread_set_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**, **thread_abort**.

Status:

Believed to be stable

task_terminate

Function — Destroys a task

SYNOPSIS

kern_return_t **task_terminate**
(task_port_t *task*);

DESCRIPTION

The **task_terminate** function kills *task* and all its threads, if any. The kernel frees all resources that are in use by the task. The kernel destroys any port for which the task holds the receive right.

PARAMETERS

task
[in task send right] The port for the task to be destroyed.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_suspend**, **task_resume**, **thread_terminate**, **thread_suspend**.

task_threads

Status:

Believed to be stable

Function — Returns a list of the threads within a task

SYNOPSIS

```
kern_return_t task_threads
                (task_port_t                    task,
                thread_port_array_t*            thread_list,
                mach_msg_type_number_t*         thread_count);
```

DESCRIPTION

The **task_threads** function returns a list of the threads within *task*. The calling task or thread also receives a send right to the kernel port for each listed thread.

PARAMETERS

task
: [in task send right] The port for the task for which the thread list is to be returned.

thread_list
: [out pointer to dynamic array of thread send rights] The returned list of threads within *task*, in no particular order.

thread_count
: [out scalar] The returned count of threads in *thread_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_create**, **thread_terminate**, **thread_suspend**.

CHAPTER 8 Host, Ledger and Clock Interface

This chapter discusses the specifics of the kernel's host interfaces. Included are functions that return status information for a host, such as kernel statistics.

Note that hosts are named both by a name port, which allows the holder to request information about the host, and a control port, which provides full control access.

Associated with each host is a (small) set of clocks used to keep track of time and to generate alarms at requested times. The kernel clock objects are also named by both a name and control port.

Status:

Believed to be stable

clock_alarm

Function — Request a clock alarm

SYNOPSIS

```
kern_return_t clock_alarm
        (clock_name_port_t      clock_name,
         alarm_type_t           alarm_type,
         tvalspec_t             alarm_time,
         mach_port_t            alarm_reply_port);
```

DESCRIPTION

The **clock_alarm** function requests that a clock send an alarm message to a specified port at a given future time. The alarm message is specified by the **clock_alarm_reply** server interface.

PARAMETERS

clock_name
: [in clock-name send right] The name (or control) port for the clock.

alarm_type
: [in scalar] How to interpret the *alarm_time* value:

  TIME_RELATIVE
  : Interpret the alarm time as relative to the current time.

  TIME_ABSOLUTE
  : Interpret the alarm time as an absolute time.

alarm_time
: [in structure] The time when the alarm is to be sent.

alarm_reply_port
: [in alarm receive (to be converted to send-once) right] A port into which the alarm message is to be sent.

NOTES

If the specified alarm time is in the past, the alarm message is sent immediately and time-stamped with the current time. Otherwise, the alarm is queued and delivered at the specified alarm time and time-stamped at that time.

The alarm will be serviced at the service time nearest the specified alarm time as governed by the current clock alarm resolution.

Not all clocks implement this service, but the REALTIME clock must. If the clock does not provide this service, this call is ignored.

RETURN VALUE

Since this is an asynchronous function that does not wait for a reply, only message transmission errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **clock_alarm_reply**.

Data Structures: **tvalspec**.

Status:

Believed to be stable

clock_alarm_reply

Server Interface — Indicates that an alarm went off

SYNOPSIS

```
kern_return_t clock_alarm_reply
        (mach_port_t        alarm_reply_port,
         kern_return_t      reply_code,
         alarm_type_t       alarm_type,
         tvalspec_t         wake_time);
```

DESCRIPTION

A **clock_alarm_reply** function is called as the result of a message from the kernel indicating that a previously requested alarm time (**clock_alarm**) has arrived.

PARAMETERS

alarm_reply_port
: [in alarm (receive) right] The reply port named in the corresponding **clock_alarm** call.

reply_code
: [in scalar] The reply status code from the alarm. The possible values are:

KERN_SUCCESS
: The alarm was delivered without problem.

KERN_INVALID_VALUE
: The *alarm_type* and/or *alarm_time* values supplied to **clock_alarm** were invalid.

KERN_INVALID_LEDGER
: The *ledger* supplied to **clock_alarm** was not a ledger.

KERN_ABORTED
: The alarm was terminated via use of **clock_set_time**.

alarm_type
: [in scalar] The alarm type value supplied to the **clock_alarm** call. Only the low order bits of this value are used by the kernel so the high order bits are available for application use as an alarm identifier.

wake_time
: [in structure] The time when the alarm message was sent.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **clock_alarm**, **clock_reply_server**, **clock_set_time**.

Data Structures: **tvalspec**.

Status:

Believed to be stable

clock_get_attributes

Function — Returns attributes of a clock

SYNOPSIS

kern_return_t **clock_get_attributes**
(clock_name_port_t *clock_name,*
clock_flavor_t *flavor,*
clock_attr_t *attribute*,
mach_msg_type_number_t* *attribute_count*);

DESCRIPTION

The **clock_get_attributes** function returns attributes of a clock's implementation or operation.

PARAMETERS

clock_name
[in clock-name send right] The name (or control) port for the clock.

flavor
[in scalar] Type of information desired. Defined values are:

CLOCK_GET_TIME_RES
The resolution, in nanoseconds, with which the value returned by **clock_get_time** is updated.

CLOCK_MAP_TIME_RES
The resolution, in nanoseconds, with which the value visible via **clock_map_time** is updated.

CLOCK_ALARM_CURRES
The resolution, in nanoseconds, at which clock alarm and sleep timers are currently serviced.

CLOCK_ALARM_MINRES
The minimum resolution, in nanoseconds, at which clock alarm and sleep timers can be serviced.

CLOCK_ALARM_MAXRES
The maximum resolution, in nanoseconds, at which clock alarm and sleep timers can be serviced.

attribute
[out scalar] The returned attribute.

attribute_count
[in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_time**, **clock_map_time**, **clock_sleep**, **clock_alarm**, **clock_set_attributes**.

clock_get_time

Status:

Believed to be stable

Function — Returns the current time

SYNOPSIS

```
kern_return_t clock_get_time
        (clock_name_port_t          clock_name,
         tvalspec_t*                cur_time);
```

DESCRIPTION

The **clock_get_time** function returns the current time kept by a clock. The value returned is a monotonically increasing value (unless tampered with via the **clock_set_time** function).

PARAMETERS

clock_name
: [in clock-name send right] The name (or control) port for the clock.

cur_time
: [out structure] Current time

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_map_time**, **clock_sleep**, **clock_alarm**, **clock_set_time**.

Data Structures: **tvalspec**.

clock_map_time

Status:

Believed to be stable

Function — Returns a memory object that maps a clock

SYNOPSIS

kern_return_t **clock_map_time**
(clock_name_port_t *clock_name,*
mem_obj_rep_port_t* *clock_memory*);

DESCRIPTION

The **clock_map_time** function returns a memory object representative port representing read access to a memory object that contains (at offset zero) a mapped version of the clock time (structure **mapped_tvalspec**). The returned right is suitable as an argument for **vm_map**.

PARAMETERS

clock_name
[in clock-name send right] The name (or control) port for the clock.

clock_memory
[out memory-object-representative send right] Mapped clock time memory object representative.

NOTES

Not all clocks provide this service, but the REALTIME clock must.

RETURN VALUE

KERN_FAILURE
The specified clock does not provide this service.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **clock_alarm**.

Data Structures: **mapped_tvalspec**.

Status:

Believed to be stable

clock_set_attributes

Function — Sets attributes of a clock

SYNOPSIS

kern_return_t **clock_set_attributes**

(clock_control_port_t	*clock_control,*
clock_flavor_t	*flavor,*
clock_attr_t	*attribute*,
mach_msg_type_number_t	*attribute_count*);

DESCRIPTION

The **clock_set_attributes** function sets attributes of a clock's operation.

PARAMETERS

clock_control
[in clock-control send right] The control port for the clock.

flavor
[in scalar] Type of information to be set. Defined values are:

CLOCK_ALARM_CURRES
The resolution, in nanoseconds, at which clock alarm and sleep timers are currently serviced. Increasing the current resolution will have no impact on any pending clock alarms (i.e. they will go off as originally scheduled). Decreasing the current resolution will truncate any pending alarms to the granularity of the new current resolution. This value must be a multiple of the minimum resolution and not greater than the maximum resolution of the clock.

attribute
[pointer to in scalar] New attribute.

attribute_count
[in scalar] The size of the buffer (in natural-sized units).

NOTES

The main reason a clock's current resolution would not always equal its minimum resolution is because the overhead of sustaining the minimum resolution, when it is not needed by any existing alarm service client, may be prohibitive for a given hardware platform and underlying clock device.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_control**, **clock_set_time**, **clock_get_attributes**.

Status:

Believed to be stable

clock_set_time

Function — Sets the current time

SYNOPSIS

kern_return_t **clock_set_time**
(clock_control_port_t *clock_control,*
tvalspec_t *new_time*);

DESCRIPTION

The **clock_set_time** function sets the time kept by a clock. Setting the clock time will cause all pending clock alarms and sleeps to be terminated with time-stamps set to the current clock time just prior to the new time being set with a return code of KERN_ABORTED.

PARAMETERS

clock_control
[in clock-control send right] The control port for the clock.

new_time
[in structure] New time

CAUTIONS

The use of this function is highly discouraged since it could affect the monotonically increasing nature of the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_control**, **clock_set_attributes**, **clock_get_time**, **clock_alarm**, **clock_sleep**.

Data Structures: **tvalspec**.

clock_sleep

System Trap — Sleep until a given time

Status:

Believed to be stable

SYNOPSIS

```
kern_return_t clock_sleep
        (clock_name_port_t        clock_name,
         sleep_type_t             sleep_type,
         tvalspec_t               sleep_time,
         tvalspec_t*              wake_time);
```

DESCRIPTION

The **clock_sleep** system trap delays the invoking thread until a specified time. This sleep may be aborted by **thread_abort**.

PARAMETERS

clock_name
[in clock-name send right] The name (or control) port for the clock.

sleep_type
[in scalar] How to interpret the *sleep_time* value:

TIME_RELATIVE
Interpret the sleep time as relative to the current time.

TIME_ABSOLUTE
Interpret the sleep time as an absolute time.

sleep_time
[in structure] The time when the sleep is to terminate.

wake_time
[out structure] The actual (absolute) time at which the sleep terminated.

NOTES

If the specified time is in the past, the call returns immediately with the wake time being the current time.

If the clock's time is changed (**clock_set_time**), the sleep will be interrupted.

The thread will waken at the service time nearest the specified sleep time as governed by the current clock alarm resolution.

Not all clocks provide this service but the REALTIME clock must.

RETURN VALUE

KERN_FAILURE
The clock does not support a sleep service.

KERN_ABORTED
The sleep was interrupted by **thread_abor**t or terminated via use of **clock_set_time**.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_alarm**, **clock_set_time**, **thread_abort**.

Data Structures: **tvalspec**.

host_get_boot_info

Status:

Believed to be stable

Function — Return operator boot information

SYNOPSIS

kern_return_t **host_get_boot_info**
    (host_control_port_t   *priv_host,*
    kernel_boot_info_t   *boot_info*);

DESCRIPTION

The **host_get_boot_info** function returns the boot-time information string supplied by the operator when *priv_host* was initialized. The constant KERNEL_BOOT_INFO_MAX (in **mach/host_info.h**) should be used to dimension storage for the returned string.

PARAMETERS

priv_host
: [in host-control send right] The control port for the host for which information is to be obtained.

boot_info
: [out array of *char*] Character string providing the operator boot info

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**.

Status:

Believed to be stable

host_get_clock_control

Function — Returns a send right to a kernel clock's control port

SYNOPSIS

```
kern_return_t host_get_clock_control
                (host_control_port_t        host_priv,
                 clock_id_t                 id,
                 clock_control_port_t*      clock_control);
```

DESCRIPTION

The **host_get_clock_control** function returns a send right to the control port for a kernel clock object. This right is used to set the clock's resolution and time.

PARAMETERS

host_priv
: [in host-control send right] The control port for the host owning the clock.

id
: [in scalar] The identification of the desired kernel clock. These values are defined in **<mach/clock_types.h>**. Although an implementation may define additional values, the following values are always defined (although only the REALTIME clock is required to be implemented):

REALTIME_CLOCK
: A moderate resolution clock service that (typically) tracks time since the system last boot.

BATTERY_CLOCK
: A (typically) low resolution clock (to the second) that survives power failures or service outages.

HIGHRES_CLOCK
: A high resolution clock.

clock_control
: [out clock-control send right] Control port for the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **clock_set_time**, **clock_set_attributes**, **host_get_clock_service**.

host_get_clock_service

Status:

Believed to be stable

Function — Returns a send right to a kernel clock's service port

SYNOPSIS

```
kern_return_t host_get_clock_service
        (host_name_port_t          host,
         clock_id_t                id,
         clock_name_port_t*        clock_name);
```

DESCRIPTION

The **host_get_clock_service** function returns a send right to the name port for a kernel clock object. This right is used to get the time and resolutions of the clock and to set clock alarms.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host owning the clock.

id
: [in scalar] The identification of the desired kernel clock. These values are defined in **<mach/clock_types.h>**. Although an implementation may define additional values, the following values are always defined (although only the REALTIME clock is required to be implemented):

REALTIME_CLOCK
: A moderate resolution clock service that (typically) tracks time since the system last boot.

BATTERY_CLOCK
: A (typically) low resolution clock (to the second) that survives power failures or service outages.

HIGHRES_CLOCK
: A high resolution clock.

clock_name
: [out clock-name send right] Name port for the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **clock_get_time**, **clock_get_attributes**, **clock_map_time**, **clock_sleep**, **clock_alarm**, **host_get_clock_control**.

host_info

Status:

Believed to be stable

Function — Returns information about a host

SYNOPSIS

kern_return_t **host_info**
    (host_name_port_t *host,*
    host_flavor_t *flavor,*
    host_info_t *host_info*,
    mach_msg_type_number_t* *host_info_count*);

DESCRIPTION

The **host_info** function returns selected information about a host, as specified by *flavor*.

PARAMETERS

host
[in host-name send right] The name (or control) port for the host for which information is to be obtained.

flavor
[in scalar] The type of statistics desired.

HOST_BASIC_INFO
Basic information (number of processors, amount of memory). The returned structure is **host_basic_info**.

HOST_SCHED_INFO
Basic restrictions of the kernel's scheduling, minimum quantum and time-out value. The returned structure is **host_sched_info**

HOST_RESOURCE_SIZES
Size of significant kernel structures, as a ledger would consider them when limiting kernel resource consumption. The returned structure is **kernel_resource_sizes**.

host_info
[out structure] Statistics about the specified host.

host_info_count
[in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

NOTES

This interface is machine word length specific because of the memory size returned by HOST_BASIC_INFO.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_boot_info**, **host_kernel_version**, **host_statistics**.

Data Structures: **host_basic_info, host_sched_info**, **kernel_resource_sizes**.

Status:

Believed to be stable

host_kernel_version

Function — Returns kernel version information for a host

SYNOPSIS

kern_return_t **host_kernel_version**
(host_name_port_t *host,*
kernel_version_t *version*);

DESCRIPTION

The **host_kernel_version** function returns the version string compiled into the kernel executing on *host* at the time it was built. This describes the version of the kernel. The constant KERNEL_VERSION_MAX (in **mach/host_info.h**) should be used to dimension storage for the returned string if the *kernel_version_t* declaration is not used.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host for which information is to be obtained.

version
: [out array of *char*] Character string describing the kernel version executing on *host*

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**.

host_reboot

Status:

Believed to be stable

Function — Reboot this host

SYNOPSIS

kern_return_t **host_reboot**
(host_control_port_t *host_priv,*
reboot_t *options*);

DESCRIPTION

The **host_reboot** function reboots the specified host.

PARAMETERS

host_priv
[in host-control send right] The control port the host to be re-booted.

options
[in scalar] Reboot options. See **<sys/reboot.h>** for details.

NOTES

If successful, this call will not return.

RETURN VALUE

Only generic errors apply.

host_statistics

Status:

Believed to be stable

Function — Returns statistics for a host

SYNOPSIS

```
kern_return_t host_statistics
        (host_control_port_t            host_priv,
         host_flavor_t                  flavor,
         host_info_t                    host_info,
         mach_msg_type_number_t*        host_info_count);
```

DESCRIPTION

The **host_statistics** function returns scheduling and virtual memory statistics concerning the host as specified by *flavor*.

PARAMETERS

host_priv
[in host-control send right] The control port for the host for which information is to be obtained.

flavor
[in scalar] The type of statistics desired.

HOST_LOAD_INFO
System loading statistics. The returned structure is **host_load_info**.

HOST_VM_INFO
Virtual memory statistics. The returned structure is **vm_statistics**.

host_info
[out structure] Statistics about the specified host.

host_info_count
[in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**, **processor_set_statistics**.

Data Structures: **host_load_info**, **vm_statistics**.

Status:

Believed to be stable

mach_host_self

System Trap — Returns the host self port

SYNOPSIS

```
host_name_port_t mach_host_self
                 ();
```

DESCRIPTION

The **mach_host_self** function returns send rights to the task's host self port. By default, this is the name port for the current host but can be a different value if so set.

PARAMETERS

None

RETURN VALUE

[host-self send right] Send rights to the host's name port.

RELATED INFORMATION

Functions: **host_info**, **task_set_special_port**.

CHAPTER 9

Processor Management and Scheduling Interface

This chapter discusses the specifics of the kernel's processor and processor set interfaces. This includes functions to control processors, change their assignments, assign tasks and threads to processors, and processor status returning functions.

Note that processor sets have two ports that name them: a name port which allows information to be requested about them, and a control port which allows full access. The control port for a processor set is provided to the creator of the set.

Processors have only a single port that names them. The host control port is needed to obtain these processor ports.

Status:

Believed to be stable

host_processor_set_priv

Function — Translates a processor set name port into a processor set control port

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t host_processor_set_priv
        (host_control_port_t                    host_priv,
         processor_set_name_port_t              set_name,
         processor_set_control_port_t*          processor_set);
```

DESCRIPTION

The **host_processor_set_priv** function returns send rights for the control port for a specified processor set currently existing on *host_priv*.

PARAMETERS

host_priv
: [in host-control send right] The control port for the host for which the processor set is desired.

set_name
: [in processor-set-name send right] The name port for the processor set desired.

processor_set
: [out processor-set-control send right] The returned processor set control port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_processor_sets, processor_set_create, processor_set_tasks, processor_set_threads**.

host_processor_sets

Status:

Believed to be stable

Function — Returns processor set ports for a host

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **host_processor_sets**
(host_control_port_t *host_priv,*
processor_set_name_port_array_t* *processor_set_name_list,*
mach_msg_type_number_t* *processor_set_name_count*);

DESCRIPTION

The **host_processor_sets** function returns send rights for the name ports for each processor set currently existing on *host*.

PARAMETERS

host_priv
[in host-control send right] The control port for the host for which the processor sets are desired.

processor_set_name_list
[out pointer to dynamic array of processor-set-name send rights] The set of processor set name ports for those currently existing on *host_priv*; no particular order is guaranteed.

processor_set_name_count
[out scalar] The number of processor set names returned.

NOTES

If control ports to the processor sets are needed, use **host_processor_set_priv**.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_processor_set_priv, processor_set_create, processor_set_tasks, processor_set_threads**.

Status:

Believed to be stable

host_processor_slots

Function — Returns the defined processor slots

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t host_processor_slots
                (host_name_port_t                    host,
                 processor_slot_t*                   slots,
                 mach_msg_type_number_t*             count);
```

DESCRIPTION

The **host_processor_slots** function returns an array of the processor slots defined for the host.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host for which information is to be obtained.

slots
: [out array of *processor_slot_t*] An array of the processor slot numbers for active processors.

count
: [pointer to in/out scalar] On input, the maximum size of the *slots* buffer; on output, the number of returned slot numbers.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info, host_processors, processor_info**.

host_processors

Status:

Believed to be stable

Function — Gets processor ports for a host

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **host_processors**

(host_control_port_t	*host_priv,*
processor_port_array_t*	*processor_list,*
mach_msg_type_number_t*	*processor_count*);

DESCRIPTION

The **host_processors** function returns an array of send right ports for each processor existing on *host_priv*.

PARAMETERS

host_priv
: [in host-control send right] The control port for the desired host.

processor_list
: [out pointer to dynamic array of processor send rights] The set of processors existing on *host_priv*; no particular order is guaranteed.

processor_count
: [out scalar] The number of ports returned in *processor_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **processor_start, processor_exit, processor_info, processor_control**, **host_processor_slots**.

Status:

Believed to be stable

processor_assign

Function — Assign a processor to a processor set

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

kern_return_t **processor_assign**

(processor_port_t	*processor,*
processor_set_control_port_t	*new_set,*
boolean_t	*wait*);

DESCRIPTION

The **processor_assign** function assigns *processor* to the set *new_set*. After the assignment is completed, the processor only executes threads that are assigned to that processor set. Any previous assignment of the processor is nullified. The master processor cannot be reassigned.

The *wait* argument indicates whether the caller should wait for the assignment to be completed or should return immediately. Dedicated kernel threads are used to perform processor assignment, so setting *wait* to FALSE allows assignment requests to be queued and performed quicker, especially if the kernel has more than one dedicated internal thread for processor assignment.

All processors take clock interrupts at all times. Redirection of other device interrupts away from processors assigned to other than the default processor set is machine dependent.

PARAMETERS

processor
: [in processor send right] The processor to be assigned.

new_set
: [in processor-set-control send right] The control port for the processor set into which the processor is to be assigned.

wait
: [in scalar] True if the call should wait for the completion of the assignment.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_create, processor_set_info, task_assign**, **thread_assign**.

Status:

Believed to be stable

processor_control

Function — Do something to a processor

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

kern_return_t **processor_control**
(processor_port_t *processor,*
processor_info_t *cmd,*
mach_msg_type_number_t *count*);

DESCRIPTION

The **processor_control** function allows privileged software to control a processor in a multi-processor that so allows it. The interpretation of *cmd* is machine dependent.

PARAMETERS

processor
[in processor send right] The processor to be controlled.

cmd
[pointer to in array of natural-sized units] An array containing the command to be applied to the processor.

count
[in scalar] The size of the *cmd* array (in natural-sized units).

NOTES

These operations are machine dependent. They may do nothing.

RETURN VALUE

KERN_FAILURE
The operation was not performed. A likely reason is that it is not supported on this processor.

RELATED INFORMATION

Functions**: processor_start, processor_exit, processor_info, host_processors**.

processor_exit

Status:

Believed to be stable

Function — Exit a processor

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_exit
                (processor_port_t                    processor);
```

DESCRIPTION

The **processor_exit** function allows privileged software to exit a processor in a multi-processor that so allows it. An exited processor is removed from the processor set to which it was assigned and ceases to be active. The interpretation of this operation is machine dependent.

PARAMETERS

processor
[in processor send right] The processor to be controlled.

NOTES

This operation is machine dependent. It may do nothing.

CAUTIONS

The ability to restart an exited processor is machine dependent.

RETURN VALUE

KERN_FAILURE
The operation was not performed. A likely reason is that it is not supported on this processor.

RELATED INFORMATION

Functions**: processor_control, processor_start, processor_info, host_processors**.

Status:

Believed to be stable

processor_get_assignment

Function — Get current assignment for a processor

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t processor_get_assignment
                (processor_port_t                    processor,
                 processor_set_name_port_t*          assigned_set);
```

DESCRIPTION

The **processor_get_assignment** function returns the name port for the processor set to which a desired processor is currently assigned.

PARAMETERS

processor
: [in processor send right] The processor whose assignment is desired.

new_set
: [out processor-set-name send right] The name port for the processor set to which *processor* is currently assigned.

RETURN VALUE

KERN_FAILURE
: *processor* is either shut down of off-line.

RELATED INFORMATION

Functions**: processor_assign, processor_set_create, processor_info, task_assign**, **thread_assign**.

processor_info

Status:

Believed to be stable

Function — Returns information about a processor.

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_info
        (processor_port_t                     processor,
         processor_flavor_t                   flavor,
         host_name_port_t*                    host,
         processor_info_t                     processor_info,
         mach_msg_type_number_t*              processor_info_count);
```

DESCRIPTION

The **processor_info** function returns selected information for a processor, as specified by *flavor*.

PARAMETERS

processor
: [in processor send right] A processor port for which information is desired.

flavor
: [in scalar] The type of information requested.

  PROCESSOR_BASIC_INFO
  : Basic information, slot number, running status, etc. The returned structure is **processor_basic_info**.

host
: [out host-name send right] The host on which the processor resides. This is the host name port.

processor_info
: [out structure] Information about the processor.

processor_info_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_start, processor_exit, processor_control, host_processors**.

Data Structures: **processor_basic_info.**

processor_set_create

Status:

Believed to be stable except for ledger reference

Function — Creates a new processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_set_create
                (host_name_port_t                    host_name,
                 processor_set_control_port_t*       new_set,
                 processor_set_name_port_t*          new_name);
```

DESCRIPTION

The **processor_set_create** function creates a new processor set and returns the two ports associated with it. The port returned in *new_set* is the control port representing the set. It is used to perform operations such as assigning processors, tasks or threads. The port returned in *new_name* is the name port which identifies the set, and is used to obtain information about the set.

PARAMETERS

host_name
: [in host-name send right] The name (or control) port for the host on which the set is to be created.

new_set
: [out processor-set-control send right] Control port used for performing operations on the new set.

new_name
: [out processor-set-name send right] Name port used to identify the new set and obtain information about it.

RETURN VALUE

KERN_INVALID_LEDGER
: *ledger* is not a wired kernel memory ledger.

RELATED INFORMATION

Functions**: processor_set_destroy, processor_set_info, processor_assign, task_assign**, **thread_assign**.

Status:

Believed to be stable

processor_set_default

Function — Returns the default processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **processor_set_default**
(host_name_port_t *host,*
processor_set_name_port_t* *default_set_name*);

DESCRIPTION

The **processor_set_default** function returns the name port for the default processor set for the specified host. The default processor set is used by all threads, tasks and processors that are not explicitly assigned to other sets.

PARAMETERS

host
[in host-name send right] The name (or control) port for the host for which the default processor set is desired.

default_set_name
[out processor-set-name send right] The returned name port for the default processor set.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_info, thread_assign, task_assign**.

processor_set_destroy

Status:

Believed to be stable

Function — Destroys a processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **processor_set_destroy**
(processor_set_control_port_t *processor_set*);

DESCRIPTION

The **processor_set_destroy** function destroys the specified processor set. Any assigned processors, tasks or threads are re-assigned to the default set. The object port (not the name port) for the processor set is required.

PARAMETERS

processor_set
[in processor-set-control send right] The control port for the processor set to be destroyed.

RETURN VALUE

KERN_DEFAULT_SET
An attempt was made to destroy the default processor set.

RELATED INFORMATION

Functions**: processor_set_create, processor_assign, task_assign**, **thread_assign**.

Status:

Believed to be stable

processor_set_info

Function — Returns information about a processor set.

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_set_info
                (processor_set_name_port_t            processor_set_name,
                 processor_set_flavor_t                           flavor,
                 host_name_port_t*                                  host,
                 processor_set_info_t                 processor_set_info,
                 mach_msg_type_number_t*        processor_set_info_count);
```

DESCRIPTION

The **processor_set_info** function returns selected information for a processor set, as specified by *flavor*.

PARAMETERS

processor_set_name
: [in processor-set-name send right] A processor set name (or control) port for which information is desired.

flavor
: [in scalar] The type of information requested.

PROCESSOR_SET_BASIC_INFO
: Basic information concerning the processor set (number of assigned processors and default policy). The returned structure is defined by **processor_set_basic_info**.

PROCESSOR_SET_TIMESHARE_DEFAULT
: The base attributes for the timeshare scheduling policy. The returned structure is **policy_timeshare_base**.

PROCESSOR_SET_FIFO_DEFAULT
: The base attributes for the FIFO scheduling policy. The returned structure is **policy_fifo_base**.

PROCESSOR_SET_RR_DEFAULT
: The base attributes for the round-robin scheduling policy. The returned structure is **policy_rr_base**.

PROCESSOR_SET_TIMESHARE_LIMITS
: Limits on the allowed timeshare policy attributes. The returned structure is defined by **policy_timeshare_limit**.

PROCESSOR_SET_RR_LIMITS
: Limits on the allowed round robin policy attributes. The returned structure is defined by **policy_rr_limit**.

PROCESSOR_SET_FIFO_LIMITS
: Limits on the allowed first-in, first-out policy attributes. The returned structure is defined by **policy_fifo_limit**.

PROCESSOR_SET_ENABLED_POLICIES
: The set of enabled policies. The returned data is a bit-vector.

host
: [out host-name send right] The name port for the host on which the processor set resides.

processor_set_info
: [out structure] Information about the processor set.

processor_set_info_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

NOTES

A processor set has a single default scheduling policy in effect for it (as returned by PROCESSOR_SET_BASIC_INFO), so only one of the default scheduling structures has valid information. On the other hand, a processor set maintains limits for all defined scheduling policies, so all of the scheduling limit structures return valid values.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_statistics**, **processor_set_create, processor_set_default**, **processor_assign**, **processor_set_policy_control**.

Data Structures: **processor_set_basic_info, policy_timeshare_info, policy_rr_info, policy_fifo_info.**

Status:

Believed to be stable

processor_set_policy_control

Function — Controls the scheduling attributes governing the processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_set_policy_control
        (processor_set_control_port_t      processor_set_control,
         processor_set_flavor_t            flavor,
         processor_set_info_t              policy_info,
         mach_msg_type_number_t            policy_info_count,
         boolean_t                         change_tasks_threads);
```

DESCRIPTION

The **processor_set_policy_control** function controls scheduling attributes governing the processor set.

PARAMETERS

processor_set_control
: [in processor-set-control send right] A processor set control port.

flavor
: [in scalar] The type of policy change to make.

PROCESSOR_SET_TIMESHARE_DEFAULT
: Change the base attributes for the timeshare scheduling policy, making timeshare the default policy. The structure is **policy_timeshare_base**.

PROCESSOR_SET_FIFO_DEFAULT
: Change the base attributes for the FIFO (first-in, first-out) scheduling policy, making FIFO the default policy. The structure is **policy_fifo_base**.

PROCESSOR_SET_RR_DEFAULT
: Changed the base attributes for the round-robin scheduling policy, making round robin the default policy. The structure is **policy_rr_base**.

PROCESSOR_SET_TIMESHARE_LIMITS
: Change the limits on the allowed timeshare policy attributes. The structure is defined by **policy_timeshare_limit**.

PROCESSOR_SET_RR_LIMITS
: Change the limits on the allowed round robin policy attributes. The structure is defined by **policy_rr_limit**.

PROCESSOR_SET_FIFO_LIMITS
: Change the limits on the allowed first-in, first-out policy attributes. The structure is defined by **policy_fifo_limit**.

PROCESSOR_SET_ENABLED_POLICIES
: Change the set of enabled policies. The data is a bit-vector.

policy_info
: [in structure] The relevant policy information.

policy_info_count
: [in scalar] The size of the buffer (in natural-sized units).

change_tasks_threads
: [in scalar] If true, any assigned task or thread whose policy is no longer enabled or whose scheduling attributes exceed the current limits will have their limits adjusted or their policy set to the default as appropriate.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_statistics**, **processor_set_create, processor_set_default**, **processor_assign**, **processor_set_info**.

Data Structures: **policy_timeshare_info, policy_rr_info, policy_fifo_info.**

Status:

Believed to be stable

processor_set_statistics

Function — Returns scheduling statistics for a processor set.

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_set_statistics
        (processor_set_control_port_t      processor_set_control,
         processor_set_flavor_t            flavor,
         processor_set_info_t              processor_set_info,
         mach_msg_type_number_t*           processor_set_info_count);
```

DESCRIPTION

The **processor_set_statistics** function returns statistics for a processor set as specified by *flavor*.

PARAMETERS

processor_set_control
: [in processor-set-control send right] A processor set control port for which information is desired.

flavor
: [in scalar] The type of information requested.

  PROCESSOR_SET_LOAD_INFO
  : Load statistics for the processor set. The returned structure is **processor_set_load_info**.

processor_set_info
: [out structure] Information about the processor set.

processor_set_info_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_info**.

Data Structures: **processor_set_load_info.**

Status:

Believed to be stable

processor_set_tasks

Function — Returns a list of tasks assigned to a processor set

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t processor_set_tasks
        (processor_set_control_port_t        processor_set,
         task_port_array_t*                  task_list,
         mach_msg_type_number_t*             task_count);
```

DESCRIPTION

The **processor_set_tasks** function returns send rights to the kernel ports for each task currently assigned to *processor_set*.

PARAMETERS

processor_set
: [in processor-set-control send right] A processor set control port for which information is desired.

task_list
: [out pointer to dynamic array of task send rights] The returned set of ports naming the tasks currently assigned to *processor_set*.

task_count
: [out scalar] The number of tasks returned in *task_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_threads, task_assign**, **thread_assign**.

processor_set_threads

Status:

Believed to be stable

Function — Returns a list of threads assigned to a processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t processor_set_threads
        (processor_set_control_port_t       processor_set,
         thread_port_array_t*               thread_list,
         mach_msg_type_number_t*            thread_count);
```

DESCRIPTION

The **processor_set_threads** function returns send rights to the kernel ports for each thread currently assigned to *processor_set*.

PARAMETERS

processor_set
: [in processor-set-control send right] A processor set control port for which information is desired.

thread_list
: [out pointer to dynamic array of thread send rights] The returned set of ports naming the threads currently assigned to *processor_set*.

thread_count
: [out scalar] The number of threads returned in *thread_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_tasks, task_assign**, **thread_assign**.

Status:

Believed to be stable

processor_start

Function — Start a processor

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **processor_start**
    (processor_port_t *processor*);

DESCRIPTION

The **processor_start** function allows privileged software to start a processor in a multi-processor that so allows it. A newly started processor is assigned to the default processor set. The interpretation of this operation is machine dependent.

PARAMETERS

processor
: [in processor send right] The processor to be controlled.

NOTES

This operation is machine dependent. It may do nothing.

CAUTIONS

The ability to restart an exited processor is machine dependent.

RETURN VALUE

KERN_FAILURE
: The operation was not performed. A likely reason is that it is not supported on this processor.

RELATED INFORMATION

Functions**: processor_control, processor_exit, processor_info, host_processors**.

task_assign

Status:

Believed to be stable

Function — Assign a task to a processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t task_assign
                (task_port_t                          task,
                 processor_set_control_port_t         processor_set,
                 boolean_t                            assign_threads);
```

DESCRIPTION

The **task_assign** function assigns *task* to the set *processor_set*. After the assignment is completed, newly created threads within this task will be assigned to this processor set. Any previous assignment of the task is nullified.

If *assign_threads* is TRUE, existing threads within the task will also be assigned to the processor set.

PARAMETERS

task
: [in task send right] The port for the task to be assigned.

processor_set
: [in processor-set-control send right] The control port for the processor set into which the task is to be assigned.

assign_threads
: [in scalar] True if this assignment should apply as well to the threads within the task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: task_assign_default, task_get_assignment, processor_set_create, processor_set_info**, **thread_assign**.

Status:

Believed to be stable

task_assign_default

Function — Assign a task to the default processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **task_assign_default**
(task_port_t *task,*
boolean_t *assign_threads*);

DESCRIPTION

The **task_assign_default** function assigns *task* to the default processor set. After the assignment is completed, newly created threads within this task will be assigned to this processor set. Any previous assignment of the task is nullified.

If *assign_threads* is TRUE, existing threads within the task will also be assigned to the processor set.

PARAMETERS

task
[in task send right] The port for the task to be assigned.

assign_threads
[in scalar] True if this assignment should apply as well to the threads within the task.

NOTES

This variant of **task_assign** exists because the control port for the default processor set is privileged, and therefore not available to most tasks.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: task_assign, task_get_assignment, processor_set_create, processor_set_info, thread_assign**.

task_get_assignment

Status:

Believed to be stable

Function — Returns the processor set to which a task is assigned

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t task_get_assignment
                (task_port_t                          task,
                 processor_set_name_port_t*           processor_set);
```

DESCRIPTION

The **task_get_assignment** function returns the name port to the processor set to which *task* is currently assigned. This port can only be used to obtain information about the processor set.

PARAMETERS

task
: [in task send right] The port for the task whose assignment is desired.

processor_set
: [out processor-set-name send right] The name port for the processor set into which the task is assigned.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: task_assign, task_assign_default, processor_set_create, processor_set_info, thread_assign**.

Status:

Believed to be stable

task_policy

Function — Sets the default scheduling attributes for a task

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

```
kern_return_t task_policy
                (task_port_t                         task,
                 policy_t                            policy,
                 policy_base_t                       base,
                 mach_msg_type_number_t              base_count,
                 boolean_t                           set_limit,
                 boolean_t                           change_threads);
```

DESCRIPTION

The **task_policy** function sets the default scheduling attributes for *task*. These attributes are used when creating new threads. Changing the default attributes for a task does not affect the attributes of the contained threads unless *change_threads* is TRUE. At no time will a thread ever have scheduling attributes that exceed the thread's limits.

PARAMETERS

task
: [in task send right] The port for the task whose scheduling attributes are to be set.

policy
: [in scalar] Default policy. The values currently defined are POLICY_-TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base
: [pointer to in structure] Base scheduling policy data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

base_count
: [in scalar] The size of the buffer (in natural-sized units).

set_limit
: [in scalar] True if the scheduling limits for the task should be restricted to allow no more service than specified by *base*.

change_threads
[in scalar] True if the attributes (and limits, if *set_limit* is true) of existing threads within the task should also be changed.

RETURN VALUE

KERN_INVALID_POLICY
The processor set does not currently enable *policy*.

KERN_POLICY_LIMIT
The specified scheduling attributes exceeds the thread's limits.

RELATED INFORMATION

Functions: **thread_policy**, **thread_set_policy**, **task_set_policy**, **processor_set_-policy_control**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

Status:

Believed to be stable

task_set_policy

Function — Sets the scheduling policy to apply to a task

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t task_set_policy
                (task_port_t                          task,
                 processor_set_control_port_t         processor_set,
                 policy_t                             policy,
                 policy_base_t                        base,
                 mach_msg_type_number_t               base_count,
                 policy_limit_t                       limit,
                 mach_msg_type_number_t               limit_count,
                 boolean_t                            change_threads);
```

DESCRIPTION

The **task_set_policy** function sets the scheduling attributes, both base and limit, for *task*. *policy* may be any policy implemented by the processor set whether or not it is enabled.

PARAMETERS

task
: [in task send right] The task whose scheduling policy is to be set.

processor_set
: [in processor-set-control send right] The control port for the processor set to which the task is currently assigned.

policy
: [in scalar] Policy to be set. The values currently defined are POLICY_-TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base
: [pointer to in structure] Base policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

base_count
: [in scalar] The size of the buffer (in natural-sized units).

limit
[pointer to in structure] Policy specific limits, **policy_fifo_limit**, **policy_rr_limit** or **policy_timeshare_limit**.

limit_count
[in scalar] The size of the buffer (in natural-sized units).

change_threads
[in scalar] True if the scheduling attributes for all contained threads should be changed as well.

RETURN VALUE

KERN_INVALID_PROCESSOR_SET
processor_set is not the task's processor set control port.

RELATED INFORMATION

Functions: **processor_set_policy_control**, **thread_policy**, **thread_set_policy**, **task_policy**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

Status:

Believed to be stable

thread_assign

Function — Assign a thread to a processor set

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t thread_assign
                (thread_port_t                          thread,
                processor_set_control_port_t            processor_set);
```

DESCRIPTION

The **thread_assign** function assigns *thread* to the set *processor_set*. After the assignment is completed, the thread executes only on processors that are assigned to that processor set. Any previous assignment of the thread is nullified.

PARAMETERS

thread
: [in thread send right] The thread to be assigned.

processor_set
: [in processor-set-control send right] The control port for the processor set into which the thread is to be assigned.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: thread_assign_default, thread_get_assignment, processor_set_create, processor_set_info, task_assign**.

thread_assign_default

Status:

Believed to be stable

Function — Assign a thread to the default processor set

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **thread_assign_default**
(thread_port_t *thread*);

DESCRIPTION

The **thread_assign_default** function assigns *thread* to the default processor set. After the assignment is completed, the thread executes only on processors that are assigned to that processor set. Any previous assignment of the thread is nullified.

PARAMETERS

thread
[in thread send right] The thread to be assigned.

NOTES

This variant of **thread_assign** exists because the control port for the default processor set is privileged, and therefore not available to most tasks.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: thread_assign, thread_get_assignment, processor_set_create, processor_set_info, task_assign**.

Status:

Believed to be stable

thread_get_assignment

Function — Returns the processor set to which a thread is assigned

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **thread_get_assignment**
(thread_port_t *thread,*
processor_set_name_t* *processor_set*);

DESCRIPTION

The **thread_get_assignment** function returns the name port to the processor set to which *thread* is currently assigned. This port can only be used to obtain information about the processor set.

PARAMETERS

thread
[in thread send right] The thread whose assignment is desired.

processor_set
[out processor-set-name send right] The name port for the processor set into which the thread is assigned.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: thread_assign, thread_assign_default, processor_set_create, processor_set_info, task_assign**.

thread_policy

Status:

Believed to be stable

Function — Sets the scheduling policy to apply to a thread

LIBRARY

#include <**mach/mach_host.h**>

SYNOPSIS

```
kern_return_t thread_policy
                (thread_port_t                    thread,
                 policy_t                         policy,
                 policy_base_t                    base,
                 mach_msg_type_number_t           base_count,
                 boolean_t                        set_limit);
```

DESCRIPTION

The **thread_policy** function sets the scheduling policy to be applied to *thread*. *policy* must be a scheduling policy currently "enabled" for the thread's assigned processor set.

PARAMETERS

thread
: [in thread send right] The thread scheduling policy is to be set.

policy
: [in scalar] Policy to be set. The values currently defined are POLICY_TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base
: [pointer to in structure] Base scheduling policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

base_count
: [in scalar] The size of the buffer (in natural-sized units).

set_limit
: [in scalar] True if the thread's scheduling limits should be restricted to allow no more service than specified by *base*.

RETURN VALUE

KERN_INVALID_POLICY
The processor set to which *thread* is currently assigned does not currently enable *policy*.

KERN_POLICY_LIMIT
The specified scheduling attributes exceeds the thread's limits.

RELATED INFORMATION

Functions: **processor_set_policy_control**, **thread_set_policy**, **task_policy**, **task_set_policy**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

thread_set_policy

Status:

Believed to be stable

Function — Sets the scheduling policy to apply to a thread

LIBRARY

#include **<mach/mach_host.h>**

SYNOPSIS

kern_return_t **thread_set_policy**	
(thread_port_t	*thread,*
processor_set_control_port_t	*processor_set,*
policy_t	*policy,*
policy_base_t	*base*,
mach_msg_type_number_t	*base_count*,
policy_limit_t	*limit*,
mach_msg_type_number_t	*limit_count*);

DESCRIPTION

The **thread_set_policy** function sets the scheduling attributes, both base and limit, for *thread*. *policy* may be any policy implemented by the processor set whether or not it is enabled.

PARAMETERS

thread
: [in thread send right] The thread scheduling policy is to be set.

processor_set
: [in processor-set-control send right] The control port for the processor set to which the thread is currently assigned.

policy
: [in scalar] Policy to be set. The values currently defined are POLICY_TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base
: [pointer to in structure] Base policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

base_count
: [in scalar] The size of the buffer (in natural-sized units).

limit
[pointer to in structure] Policy specific limits, **policy_fifo_limit**, **policy_rr_limit** or **policy_timeshare_limit**.

limit_count
[in scalar] The size of the buffer (in natural-sized units).

RETURN VALUE

KERN_INVALID_PROCESSOR_SET
processor_set is not the thread's processor set control port.

RELATED INFORMATION

Functions: **processor_set_policy_control**, **thread_policy**, **task_policy**, **task_set_policy**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

CHAPTER 10 In-Kernel Device Interface

This chapter discusses the specifics of the device interfaces to in-kernel device drivers. These interfaces provide read, write and status interfaces to devices.

A new set of interfaces have been added to provide for asynchronous I/O support which allows for completion notification directly from the device driver interrupt context.

Status:

Believed to be stable

device_close

Function — De-establish a connection to a device.

LIBRARY

#include <**device/device.h**>

SYNOPSIS

```
kern_return_t device_close
        (device_port_t                                  device);
```

DESCRIPTION

The **device_close** function destroys the associated device port. The open count for the named device is decremented. If this count reaches zero, the close operation of the device driver is invoked, closing the device.

PARAMETERS

device
: [in device send right] A device port to the device to be closed.

NOTES

device_close will destroy any mapped device windows obtained through this device port.

RETURN VALUE

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

RELATED INFORMATION

Functions: **device_open**.

device_get_status

Status:

Believed to be stable

The MDIOCGETL flavor is experimental. **[OSF-RI DIPC]**

Function — Return the current device status

LIBRARY

#include **<device/device.h>**
#include **<device/ds_status.h>**

SYNOPSIS

kern_return_t **device_get_status**

(device_port_t	*device,*
device_flavor_t	*flavor,*
dev_status_t	*status,*
mach_msg_type_number_t*	*status_count*);

DESCRIPTION

The **device_get_status** function returns status information pertaining to an open device. The possible values for *flavor* as well as the meaning of the returned status information is device dependent, except for MDIOCGETL which is generic.

PARAMETERS

device
: [in device send right] A device port to the device to be interrogated.

flavor
: [in scalar] The type of status information requested. The only device generic flavor is:

  MDIOCGETL
  Returns the current value of the device's I/O operations limit (i.e. the number of outstanding requests on the device).

status
: [out array of natural-sized units] The returned device status.

status_count
: [pointer to in/out scalar] On input, the reserved size of *status*; on output, the size of the returned device status (in natural-sized units).

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

D_OUT_OF_BAND
Out-of-band condition occurred on device (such as typing control-C)

RELATED INFORMATION

Functions: **device_set_status**.

device_map

Status:

Believed to be stable

Function — Establish a memory manager representing a device

LIBRARY

#include **<device/device.h>**

SYNOPSIS

```
kern_return_t device_map
        (device_port_t           device,
         vm_prot_t               prot,
         vm_size_t               size,
         mem_obj_rep_port_t*     pager);
```

DESCRIPTION

The **device_map** function establishes a memory manager that presents a memory object representing a device. The resulting port is suitable for use as the memory manager port in a **vm_map** call. This call is device dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be mapped.

prot
: [in scalar] Protection for the device memory.

size
: [in scalar] The size of the device memory object.

pager
: [out memory-object-representative send right] The returned memory object representative port to a memory manager that represents the device.

NOTES

Port rights are maintained as follows:

Memory object representative port (and abstract memory object port):
: The device memory manager has all rights.

Memory cache control port:
: The device memory manager has only send rights.

Regardless how the object is created, the control port is created by the kernel and passed through the memory management interface.

If the device port used is restricted to use by a single identity, the generated representative port will be likewise restricted.

CAUTIONS

The device memory manager assumes that access to its memory objects will not be propagated to more that one host, and therefore provides no consistency guarantees beyond those made by the kernel.

In the event that more than one host attempts to use a device memory object, the device memory manager will only record the last set of port names. Currently, the device memory manager assumes that its clients adhere to the initialization and termination protocols in the memory management interface; otherwise, port rights or out-of-line memory from erroneous messages may be allowed to accumulate.

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_READ_ONLY
: Data cannot be written to this device.

RELATED INFORMATION

Functions: **vm_map**.

device_open

Status:

Believed to be stable

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
        (device_master_port_t        master_port,
         dev_mode_t                  mode,
         security_token_t            security_id,
         dev_name_t                  name,
         device_port_t*              device);
```

device_open_request

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

Asynchronous Function form — Asynchronously request a connection to a device

```
kern_return_t device_open_request
        (device_master_port_t        master_port,
         mach_port_t                 reply_port,
         dev_mode_t                  mode,
         security_token_t            security_token,
         dev_name_t                  name);
```

ds_device_open_reply

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

Asynchronous Server Interface form — Receive the reply from an asynchronous open

```
kern_return_t ds_device_open_reply
        (mach_port_t                 reply_port,
         kern_return_t               return_code,
         device_port_t               device);
```

DESCRIPTION

The **device_open** function opens a device object. The open operation of the device is invoked, if the device is not already open. The open count for the device is incremented. Each open for a device returns a distinct port, the allowed operations upon which being governed by *mode*.

PARAMETERS

master_port
: [in device-master send right] The master device port. This port is provided to the bootstrap task.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which a reply is to be sent when the device is open.

mode
: [in scalar] Opening mode. This is the bit-wise OR of the following values:

  D_READ
  : Read access

  D_WRITE
  : Write access

  D_NODELAY
  : Do not delay on open

security_id
: [in scalar] The security ID that tasks attempting to use this device port must have. A zero value indicates all identities.

name
: [pointer to in array of *char*] Name of the device to open.

return_code
: [in scalar] Status of the open.

device
: [out device send right, in for asynchronous form] The returned device port.

RETURN VALUE

device_open_request returns only message transmission errors. The return value supplied to **ds_device_open_reply** is irrelevant. The *return_code* returned by **ds_device_open_reply** or the error return from **device_open** is one of the following:

D_WOULD_BLOCK
: The device is busy, but D_NOWAIT was specified in *mode*.

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_DEVICE_DOWN
: The device has been shut down.

D_READ_ONLY
: Data cannot be written to this device.

RELATED INFORMATION

Functions: **device_close**, **device_reply_server**.

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

device_read

Function — Read a sequence of bytes from a device object.

LIBRARY

#include <**device/device.h**> (**device_read**)

#include <**device/device_request.h**> (**device_read_request**)

#include <**device/device_reply.h**> (**ds_device_read_reply**)

SYNOPSIS

kern_return_t **device_read**
(device_port_t *device,*
dev_mode_t *mode,*
recnum_t *recnum,*
io_buf_len_t *bytes_wanted,*
kern_return_t* *device_code*,
pointer_t* *data,*
mach_msg_type_number_t* *data_count*);

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

device_read_request

Asynchronous Function form — Asynchronously read data

kern_return_t **device_read_request**
(device_port_t *device,*
mach_port_t *reply_port,*
dev_mode_t *mode,*
recnum_t *recnum,*
io_buf_len_t *bytes_wanted*);

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

ds_device_read_reply

Asynchronous Server Interface form — Receive the reply from an asynchronous read

kern_return_t **ds_device_read_reply**
(mach_port_t *reply_port,*
kern_return_t *return_code,*
kern_return_t *device_code*,
pointer_t *data,*
mach_msg_type_number_t *data_count*);

DESCRIPTION

The **device_read** function reads a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

The **device_read** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

return_code
: [in scalar] The return status code from the read.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

  D_DEVICE_DOWN
  : Device has been shut down

  D_INVALID_RECNUM
  : Invalid record (block) number

  D_INVALID_SIZE
  : Invalid IO size

  D_IO_ERROR
  : Hardware IO error

  D_NO_SUCH_DEVICE
  : No device with that name, or the device is not operational.

D_OUT_OF_BAND
Out-of-band condition occurred on device (such as typing control-C)

D_WOULD_BLOCK
Operation would block, but D_NOWAIT set

data
[out pointer to dynamic array of bytes, in for asynchronous form] Returned data bytes.

data_count
[out scalar, in for asynchronous form] Number of returned data bytes.

RETURN VALUE

device_read_request returns only message transmission errors. A return value supplied to **ds_device_read_reply** other than KERN_SUCCESS or MIG_NO_REPLY will cause **mach_msg_server** to de-allocate the returned data. The return value from **device_read** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_read_inband**, **device_read_overwrite**, **device_reply_server**.

device_read_async

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

Function — Read a sequence of bytes from a device object.

LIBRARY

#include **<device/device.h>** (**device_read_async**)

SYNOPSIS

```
kern_return_t device_read_async
                (device_port_t                    device,
                mach_port_t                       queue;
                mach_port_t                       request_id;
                dev_mode_t                        mode,
                recnum_t                          recnum,
                io_buf_len_t                      bytes_wanted);
```

DESCRIPTION

The **device_read_async** function enqueues a read operation for a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_queue_create**.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_read_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**.

device_read_inband

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

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**

(device_port_t	*device,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted,*
kern_return_t*	*device_code*,
pointer_t	*data,*
mach_msg_type_number_t*	*data_count*);

device_read_request_inband

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

Asynchronous Function form — Asynchronously read data

kern_return_t **device_read_request_inband**

(device_port_t	*device,*
mach_port_t	*reply_port,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted*);

ds_device_read_reply_inband

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

Asynchronous Server Interface form — 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,*
kern_return_t	*device_code*,
pointer_t	*data,*
mach_msg_type_number_t	*data_count*);

DESCRIPTION

The **device_read** function reads a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_read** in that the returned bytes are returned “inband” in the reply IPC message.

The **device_read_inband** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

    D_NOWAIT
    : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

return_code
: [in scalar] The return status code from the read.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

    D_DEVICE_DOWN
    : Device has been shut down

    D_INVALID_RECNUM
    : Invalid record (block) number

    D_INVALID_SIZE
    : Invalid IO size

    D_IO_ERROR
    : Hardware IO error

    D_NO_SUCH_DEVICE
    : No device with that name, or the device is not operational.

D_OUT_OF_BAND
: Out-of-band condition occurred on device (such as typing control-C)

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set

data
: [out array of bytes, in for asynchronous form] Returned data bytes.

data_count
: [out scalar, in for asynchronous form] Number of returned data bytes.

RETURN VALUE

device_read_request_inband returns only message transmission errors. The return value supplied to **ds_device_read_reply_inband** is irrelevant. The return value from **device_read_inband** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_read**, **device_read_overwrite, device_reply_server**.

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

device_read_async_inband

Function — Read a sequence of bytes “inband” from a device object.

LIBRARY

#include <**device/device.h**> (**device_read_async_inband**)

SYNOPSIS

kern_return_t **device_read_async_inband**

(device_port_t	*device,*
mach_port_t	*queue;*
mach_port_t	*request_id;*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted)*;

DESCRIPTION

The **device_read_async_inband** function enqueues a read operation for a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_read_async** in that the returned bytes are returned “inband” in the completion IPC message (in io_done_result.qd_inline).

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_-queue_create**.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
[in scalar] Size of data transfer.

RETURN VALUE

device_read_async_inband returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**.

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

device_read_overwrite

System Trap — Read a sequence of bytes from a device object into the caller's address space

LIBRARY

#include **<device/device.h>** (**device_read_overwrite**)

#include **<device/device_request.h>** (**device_read_request_overwrite**)

#include **<device/device_reply.h>** (**ds_device_read_reply_overwrite**)

SYNOPSIS

kern_return_t **device_read_overwrite**

(device_port_t	*device,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted,*
pointer_t	*data,*
kern_return_t*	*device_code*,
io_buf_len_t*	*data_count*);

Status:

To Be Deleted in favorof new asynchronous support
[OSF-RI MK6]

device_read_overwrite_request

Asynchronous System Trap form — Asynchronously read data into the caller's buffer

kern_return_t **device_read_overwrite_request**

(device_port_t	*device,*
mach_port_t	*reply_port,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted*,
pointer_t	*data*);

Status:

To Be Deleted in favorof new asynchronous support
[OSF-RI MK6]

ds_device_read_reply_overwrite

Asynchronous Server Interface form — Receive the reply from an asynchronous read

kern_return_t **ds_device_read_reply_overwrite**

(mach_port_t	*reply_port,*
kern_return_t	*return_code,*
kern_return_t	*device_code*,
io_buf_len_t	*data_count*);

DESCRIPTION

The **device_read_overwrite** system trap reads a sequence of bytes from a device object directly into the caller's address space. The meaning of *recnum* as well as the specific operation performed is device dependent.

The **device_read_overwrite** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

return_code
: [in scalar] The return status code from the read.

data
: [pointer to in array of bytes] Data buffer to be overwritten.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_-code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

  D_DEVICE_DOWN
  : Device has been shut down

  D_INVALID_RECNUM
  : Invalid record (block) number

  D_INVALID_SIZE
  : Invalid IO size

  D_IO_ERROR
  : Hardware IO error

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_OUT_OF_BAND
: Out-of-band condition occurred on device (such as typing control-C)

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set

data_count
: [out scalar, in for asynchronous form] Number of returned data bytes.

RETURN VALUE

device_read_request_overwrite returns only invalid parameter errors. The return value supplied to **ds_device_read_reply_overwrite** is irrelevant. The return value from **device_read_overwrite** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_read**, **device_read_inband**, **device_reply_server**.

device_read_overwrite_async

System Trap— Read a sequence of bytes from a device object into the caller's address space.

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

LIBRARY

#include **<device/device.h>** (**device_read_overwrite_async**)

SYNOPSIS

kern_return_t **device_read_overwrite_async**

(device_port_t	*device,*
mach_port_t	*queue;*
mach_port_t	*request_id;*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted;*
io_buf_ptr_t	*buffer)*;

DESCRIPTION

The **device_read_overwrite** system trap enqueues a read operation for a sequence of bytes from a device object to be placed directly into the caller's address space. The meaning of *recnum* as well as the specific operation performed is device dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from the **io_done_-queue_create** call.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
[in scalar] Size of data transfer.

buffer
[pointer to in array of bytes] Data buffer to be overwritten.

RETURN VALUE

device_read_overwrite_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_-write_async**, **device_write_async_inband**, **io_done_queue_create**.

device_set_filter

Status:

Believed to be stable

Function — Names an input filter for a device

LIBRARY

#include <**device/device.h**>

#include <**device/net_status.h**>

SYNOPSIS

kern_return_t **device_set_filter**

(device_port_t	*device,*
filter_port_t	*receive_port,*
mach_msg_type_name_t	*receive_port_type,*
net_priority_t	*priority,*
filter_array_t	*filter,*
mach_msg_type_number_t	*filter_count*);

DESCRIPTION

The **device_set_filter** function provides a means by which selected data appearing at a device interface can be selected and routed to a port. This service is provided in support of network devices.

The filter command list consists of an array of up to NET_MAX_FILTER (16-bit) values to be applied to incoming data "messages" to determine if that data should be given to a particular input filter.

Each filter command list specifies a sequence of actions which leave a boolean value on the top of an internal stack. Each 16-bit value of the command list specifies a data (push) operation (high order NETF_NBPO bits) as well as a binary operator (low order NETF_NBPA bits).

The value to be pushed onto the stack is chosen as follows.

NETF_PUSHLIT
: Use the next 16-bit value of the filter as the value.

NETF_PUSHZERO
: Use 0 as the value.

NETF_PUSHWORD+*N*
: Use 16-bit value *N* of the "data" portion of the message as the value.

NETF_PUSHHDR+*N*
: Use 16-bit value *N* of the "header" portion of the message as the value.

NETF_PUSHIND
: Pops the top 32-bit value from the stack and then uses it to index the 16-bit value of the “data” portion of the message to be used as the value.

NETF_PUSHHDRIND
: Pops the top 32-bit value from the stack and then uses it to index the 16-bit value of the “header” portion of the message to be used as the value.

NETF_PUSHSTK+*N*
: Use 32-bit value *N* of the stack (where the top of stack is value 0) as the value.

NETF_NOPUSH
: Don’t push a value.

The unsigned value so chosen is promoted to a 32-bit value before being pushed.

Once a value is pushed (except for the case of NETF_NOPUSH), the top two 32-bit values of the stack are popped and a binary operator applied to them (with the old top of stack as the second operand). The result of the operator is pushed on the stack. These operators are:

NETF_NOP
: Don’t pop off any values and do no operation.

NETF_EQ
: Perform an equal comparison.

NETF_LT
: Perform a less than comparison.

NETF_LE
: Perform a less than or equal comparison.

NETF_GT
: Perform a greater than comparison.

NETF_GE
: Perform a greater than or equal comparison.

NETF_AND
: Perform a bit-wise boolean AND operation.

NETF_OR
: Perform a bit-wise boolean inclusive OR operation.

NETF_XOR
: Perform a bit-wise boolean exclusive OR operation.

NETF_NEQ
: Perform a not equal comparison.

NETF_LSH
: Perform a left shift operation.

NETF_RSH
: Perform a right shift operation.

NETF_ADD
: Perform an addition.

NETF_SUB
: Perform a subtraction.

NETF_COR
: Perform an equal comparison. If the comparison is TRUE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CAND
: Perform an equal comparison. If the comparison is FALSE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CNOR
: Perform a not equal comparison. If the comparison is FALSE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CNAND
: Perform a not equal comparison. If the comparison is TRUE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

The scan of the filter list terminates when the filter list is emptied, or a NETF_C... operation terminates the list. At this time, if the final value of the top of the stack is TRUE, then the message is accepted for the filter.

PARAMETERS

device
: [in device send right] A device port

receive_port
: [in filter send or receive (to be converted to send) right] The port to receive the input data that is selected by the filter.

receive_port_type

[in scalar] IPC type of the send right provided to the device; either MACH_MSG_TYPE_MAKE_SEND, MACH_MSG_TYPE_MOVE_-SEND or MACH_MSG_TYPE_COPY_SEND.

priority

[in scalar] Used to order multiple receivers in an implementation dependent way.

filter

[pointer to in array of *filter_t*] The address of an array of filter values.

filter_count

[in scalar] The size of the *filter* array (in 16-bit values).

NOTES

The kernel implementation of a network device is free to drop packets. For example, a network device will drop a message (i.e. a packet) if the attempt to send the message encounters a full queue. This is necessary to avoid flooding receivers with packets.

RETURN VALUE

D_DEVICE_DOWN

Device has been shut down

D_INVALID_OPERATION

No filter port was supplied.

D_NO_SUCH_DEVICE

No device with that name, or the device is not operational.

RELATED INFORMATION

Functions: **receive_packet**.

device_set_status

Status:

Believed to be stable

The MDIOCSETL flavor is experimental. **[OSF-RI DIPC]**

Function — Sets device status.

LIBRARY

#include <**device/device.h**>
#include <**device/ds_status.h**>

SYNOPSIS

```
kern_return_t device_set_status
        (device_port_t                    device,
         device_flavor_t                  flavor,
         dev_status_t                     status,
         mach_msg_type_number_t           status_count);
```

DESCRIPTION

The **device_set_status** function sets device status. The possible values of *flavor* as well as the corresponding meanings are device dependent, except for MDIOCSETL which is generic.

PARAMETERS

device
: [in device send right] A device port to the device to be manipulated.

flavor
: [in scalar] The type of status information to set. The only device generic flavor is:

    MDIOCSETL
    Returns the current value of the device's I/O operations limit (i.e. the number of outstanding requests on the device).The new value must be greater than or equal to ZERO and less than IO_IN_PROGRESS_MAX.

status
: [pointer to in array of natural-sized units] The status information to set.

status_count
: [in scalar] The size of the status information (in natural-sized units).

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_INVALID OPERATION
: The proposed MDIOCSETL value was not in the range 0 through IO_-IN_PROGRESS_MAX.

D_IO_ERROR
: Hardware IO error

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_OUT_OF_BAND
: Out-of-band condition occurred on device (such as typing control-C)

D_READ_ONLY
: Data cannot be written to this device.

RELATED INFORMATION

Functions: **device_get_status**.

device_write

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

Function — Write a sequence of bytes to a device object.

LIBRARY

#include **<device/device.h>** (**device_write**)

#include **<device/device_request.h>** (**device_write_request**)

#include **<device/device_reply.h>** (**ds_device_write_reply**)

SYNOPSIS

```
kern_return_t device_write
                (device_port_t                    device,
                 dev_mode_t                       mode,
                 recnum_t                         recnum,
                 pointer_t                        data,
                 mach_msg_type_number_t           data_count,
                 boolean_t                        deallocate,
                 kern_return_t*                   device_code,
                 io_buf_len_t*                    bytes_written);
```

device_write_request

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

Asynchronous Function form — Asynchronously write data

```
kern_return_t device_write_request
                (device_port_t                    device,
                 mach_port_t                      reply_port,
                 dev_mode_t                       mode,
                 recnum_t                         recnum,
                 pointer_t                        data,
                 mach_msg_type_number_t           data_count,
                 boolean_t                        deallocate);
```

ds_device_write_reply

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

Asynchronous Server Interface form — Receive the reply from an asynchronous write

```
kern_return_t ds_device_write_reply
                (mach_port_t                      reply_port,
                 kern_return_t                    return_code,
                 kern_return_t                    device_code,
                 io_buf_len_t                     bytes_written);
```

DESCRIPTION

The **device_write** function writes a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

The **device_write** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be written.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait for I/O completion.

recnum
: [in scalar] Record number to be written.

data
: [pointer to in array of bytes] Data bytes to be written.

data_count
: [in scalar] Number of data bytes to be written.

deallocate
: [in scalar] If TRUE, the pages to be written will be de-allocated from the caller's address space.

return_code
: [in scalar] The return status code from the write.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_-code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

  D_DEVICE_DOWN
  : Device has been shut down

  D_INVALID_RECNUM
  : Invalid record (block) number

D_INVALID_SIZE
: Invalid IO size

D_IO_ERROR
: Hardware IO error

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_OUT_OF_BAND
: Out-of-band condition occurred on device (such as typing control-C)

D_READ_ONLY
: Data cannot be written to this device.

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set

bytes_written
: [out scalar, in for asynchronous form] Size of data transfer.

RETURN VALUE

device_write_request returns only message transmission errors. The return value supplied to **ds_device_write_reply** is irrelevant. The return value from **device_write** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_write_inband**, **device_reply_server**.

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

device_write_async

Function — Write a sequence of bytes to a device object.

LIBRARY

#include <**device/device.h**> (**device_write_async**)

SYNOPSIS

```
kern_return_t device_write_async
        (device_port_t       device,
         mach_port_t         queue;
         mach_port_t         request_id;
         dev_mode_t          mode,
         recnum_t            recnum,
         io_buf_ptr_t        data,
         io_buf_len_t        bytes_wanted);
```

DESCRIPTION

The **device_write_async** function enqueues a write operation for a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_queue_create**.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

data
: [pointer to in array of bytes] Data bytes to be written.

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_write_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async_inband, device_read_overwrite_async, device_write_async_inband, io_done_queue_create**.

Status:

Believed to be stable

Use of the I/O operations limit is experimental.
[OSF-RI DIPC]

device_write_inband

Function — Write a sequence of bytes "inband" to a device object.

LIBRARY

#include **<device/device.h>** (**device_write_inband**)

#include **<device/device_request.h>** (**device_write_request_inband**)

#include **<device/device_reply.h>** (**ds_device_write_reply_inband**)

SYNOPSIS

```
kern_return_t device_write_inband
        (device_port_t                 device,
         dev_mode_t                    mode,
         recnum_t                      recnum,
         pointer_t                     data,
         mach_msg_type_number_t        data_count,
         kern_return_t*                device_code,
         io_buf_len_t*                 bytes_written);
```

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

device_write_request_inband

Asynchronous Function form — Asynchronously write data

```
kern_return_t device_write_request_inband
        (device_port_t                 device,
         mach_port_t                   reply_port,
         dev_mode_t                    mode,
         recnum_t                      recnum,
         pointer_t                     data,
         mach_msg_type_number_t        data_count);
```

Status:

To Be Deleted in favorof new asynchronous support [OSF-RI MK6]

ds_device_write_reply_inband

Asynchronous Server Interface form — Receive the reply from an asynchronous write

```
kern_return_t ds_device_write_reply_inband
        (mach_port_t                   reply_port,
         kern_return_t                 return_code,
         kern_return_t                 device_code,
         io_buf_len_t                  bytes_written);
```

DESCRIPTION

The **device_write_inband** function writes a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_write** in that the bytes to be written are sent "inband" in the request IPC message.

The **device_write_inband** function will block until the current number of operations on the device is less than the device's I/O operations limit.

PARAMETERS

device
: [in device send right] A device port to the device to be written.

reply_port
: [in reply receive (to be converted to send-once) right] The port to which the reply message is to be sent.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait for I/O completion.

recnum
: [in scalar] Record number to be written.

data
: [pointer to in array of bytes] Data bytes to be written.

data_count
: [in scalar] Number of data bytes to be written.

return_code
: [in scalar] The return status code from the write.

device_code
: [out scalar, in for asynchronous form] Device completion status. If the return value from the call (*return_code* in the asynchronous case) is not KERN_SUCCESS, the operation failed outright. If the operation partially succeeded, the return value is KERN_SUCCESS and *device_-code* reflects the status of the partial completion, D_SUCCESS if the operation was completely successful.

  D_DEVICE_DOWN
  : Device has been shut down

  D_INVALID_RECNUM
  : Invalid record (block) number

  D_INVALID_SIZE
  : Invalid IO size

  D_IO_ERROR
  : Hardware IO error

D_NO_SUCH_DEVICE
: No device with that name, or the device is not operational.

D_OUT_OF_BAND
: Out-of-band condition occurred on device (such as typing control-C)

D_READ_ONLY
: Data cannot be written to this device.

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set

bytes_written
: [out scalar, in for asynchronous form] Size of data transfer.

RETURN VALUE

device_write_request_inband returns only message transmission errors. The return value supplied to **ds_device_write_reply_inband** is irrelevant. The return value from **device_write_inband** (or *return_code*) can be a device error code if that error caused the operation to fail completely.

RELATED INFORMATION

Functions: **device_write**, **device_reply_server**.

device_write_async_inband

Function — Write a sequence of bytes “inband” to a device object.

Status:

New: [OSF-RI MK6] Added to support asynchronous I/O completion from interrupt context.

LIBRARY

#include **<device/device.h>** (**device_write_async_inband**)

SYNOPSIS

kern_return_t **device_write_async_inband**

(device_port_t	*device,*
mach_port_t	*queue;*
mach_port_t	*request_id;*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_ptr_t	*data,*
io_buf_len_t	*bytes_wanted)*;

DESCRIPTION

The **device_write_async_inband** function enqueues a write operation for a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device dependent. This call differs from **device_write_async** in that the bytes to be written are sent “inband” in the request IPC message.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_queue_create**.

request_id
: [in send right] An unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

data
[pointer to in array of bytes] Data bytes to be written.

bytes_wanted
[in scalar] Size of data transfer.

RETURN VALUE

device_write_async_inband returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async_inband, device_read_overwrite_async, device_write_async, io_done_queue_create**.

io_done_queue_create

Status:

New: [OSF-RI MK6] Function added to support asynchronous I/O from interrupt context.

Function — Create an io_done_queue kernel object

SYNOPSIS

```
kern_return_t io_done_queue_create
                (host_name_port_t          host,
                mach_port_t*               queue);
```

DESCRIPTION

The **io_done_queue_create** function is called to create a new instatiation of the kernel object supporting asynchronous read/write operations on a device.

PARAMETERS

host
: [in host-name send right] The name (or control) port for the host on which the io_done_queue should be created.

queue
: [out io-done-queue send right] The port referencing the created io_done_queue.

RETURN VALUE

???

RELATED INFORMATION

Functions: **io_done_queue_terminate**, **io_done_queue_wait**, **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**.

io_done_queue_terminate

Status:

New: [OSF-RI MK6] Function added to support asynchronous I/O from interrupt context.

Function — Terminate an io_done_queue kernel object

SYNOPSIS

kern_return_t **io_done_queue_terminate**
(mach_port_t* *queue*);

DESCRIPTION

The **io_done_queue_terminate** function is called to destroy a previous instatiation of the kernel object supporting asynchronous read/write operations on a device.

PARAMETERS

queue
: [in io-done-queue send right] The port referencing the io_done_queue to be destroyed.

RETURN VALUE

???

RELATED INFORMATION

Functions: **io_done_queue_create**, **io_done_queue_wait**, **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**.

io_done_queue_wait

Status:

New: [OSF-RI MK6] Function added to support asynchronous I/O from interrupt context.

Function — Wait on an io_done_queue kernel object

SYNOPSIS

kern_return_t **io_done_queue_wait**
(mach_port_t* *queue*
io_done_result_t* *result*);

DESCRIPTION

The **io_done_queue_wait** function is called to obtain the results of a previously requested asynchronous I/O operation. For each **io_done_queue_wait** invocation, the status of one I/O request is returned. If there are no pending I/O completions, **io_done_queue_wait** blocks in the kernel on the address of the completion queue. The µKernel, from interrupt context, enqueues (in FIFO order) completions (struct io_done_result's) on the completion queue and posts a wakeup on the queue for each I/O completion. Completion processing previously done by the µKernel io_done thread is now done by the task thread when it awakens.

PARAMETERS

queue
: [in io-done-queue send right] The port referencing the io_done_queue to be destroyed.

result
: [out structure] The data structure to be filled in with the completion status of the I/O operation.

RETURN VALUE

???

RELATED INFORMATION

Functions: **io_done_queue_create**, **io_done_queue_wait**, **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**.

Data Structures: **io_done_result**.

Status:

Believed to be stable

receive_packet

Server Interface — Handles the arrival of a device packet

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t receive_packet
                (filter_port_t       filter_port,
                 char*               header,
                 char*               packet);
```

DESCRIPTION

A **receive_packet** function is called by **device_server** as the result of a kernel message indicating that a packet meeting the criteria of a device's packet filter has been received.

Note that there are additional, related, interfaces that are generated by MIG and documented in the server writer's guide(e.g. maximum size of a received message, how to invoke receive unmarshalling stub, etc.)

PARAMETERS

filter_port
: [in filter (receive) right] The port to which the packet message was sent.

header
: [pointer to in array of char] The packet header, of size NET_HDW_HDR_MAX (declared in **<device/device.h>**).

packet
: [pointer to in array of char] The packet body, of size NET_RCV_MAX (declared in **<device/device.h>**).

NOTES

The kernel implementation of a network device is free to drop packets. For example, a network device will drop a message (i.e. a packet) if the attempt to send the message encounters a full queue. This is necessary to avoid flooding receivers with packets.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **device_set_filter**, **device_reply_server.**

APPENDIX A

Network Data Representation

The definition of Mach IPC permits for arbitrary encoding of the message data values into the message body and out-of-line areas. Direct users of the **mach_msg** service are free to encode data as desired.

The MIG stub generator encodes data values according to the Network Data Representation format (NDR). Since the kernel is a recipient of messages and contains MIG stubs to decode them (and to encode messages and replies it itself sends), NDR can be considered to be the kernel's data encoding. This encoding is described in this appendix.

Data Format Overview

Data generated and interpreted by MIG stubs is encoded in the message body and possibly out-of-line regions. These data areas can be viewed as a single data stream with the out-of-line region contents inserted into the message body data at the appropriate points.

The data stream begins with a format label whose format is machine independent which can be used to determine format conversions to be performed on the decoded data. This label doesn't actually describe the types of the data in the message; it simply says what machine format was used for the values that are encoded.

Following the format label are the encoded data values. Scalar data values are always directly placed into the data stream with no conversion (it is the receiver's responsibility to transform the formats if necessary). Structured values are ordered according to a prescribed set of machine independent rules, sometimes with header information describing the value's size, for example.

Out-of-line memory is used to transmit large data areas. The order of these out-of-line regions in the message is dictated by the MIG stub. No additional information is present in the data stream to describe these regions (since arrays or structures of out-of-line regions are not currently supported by MIG).

The order of port rights in the message body are also dictated by the MIG stub. For scalar rights, no additional information appears in the data stream. For port arrays, however, the information concerning the size of the array (if variable) is encoded in the data stream just as if the data were to appear there.

The Network Data Representation format specification is capable of representing more complicated types than MIG can describe. This appendix describes only that subset of NDR necessary to describe MIG types.

Data Representation Format Label

The first eight bytes of the data stream is the data representation format label used by the receiver to convert the scalar formats in the stream from the sender's representation into that of the receiver.

```
struct NDR_record_t
{
        unsigned char           mig_vers;
        unsigned char           if_vers;
        unsigned char           reserved1;
        unsigned char           mig_encoding;
        unsigned char           int_rep;
        unsigned char           char_rep;
        unsigned char           float_rep;
        unsigned char           reserved2;
};
```

mig_vers
    Currently zero.

if_vers
    Currently zero.

mig_encoding
    Format of MIG encoding, currently NDR_PROTOCOL_2_0.

int_rep
    The way in which the sender orders the bytes in an integer value.

    NDR_INT_BIG_ENDIAN
        Big endian

    NDR_INT_LITTLE_ENDIAN
        Little endian

char_rep

The major encoding of single byte characters.

NDR_CHAR_ASCII
: ASCII

NDR_CHAR_EBCDIC
: EBCDIC

float_rep

The format family used by the sender for floating point values.

NDR_FLOAT_IEEE
: IEEE (single and double format)

NDR_FLOAT_VAX
: VAX (F and G format)

NDR_FLOAT_CRAY
: Cray (IEEE single and Cray double format)

NDR_FLOAT_IBM
: IBM (short and long format)

Scalar Type Format

A scalar value is a single value as opposed to a structure or array value. The following sections describe the way in which the scalar values that compose structure or array types are ordered in the data stream. In one sense, the data stream consists of a stream of scalar values, some of which are used to describe other values in the stream.

All scalar values are transmitted in the format used by the message sender. All values must be naturally aligned: a value *N* bytes wide must start at a 0 mod *N* byte stream index (considering the first byte in the stream to be index 0). If necessary, (the smallest number of) additional bytes are added to the stream (of arbitrary value) to preserve this alignment (unless the next value would be out-of-line, in which case the out-of-line region must have the proper alignment).

Boolean

The MIG fundamental type MACH_MSG_TYPE_BOOLEAN represents a single boolean value. This value is encoded as a single byte:

- 0—False
- non-zero—True

Character

The MIG fundamental type MACH_MSG_TYPE_CHAR represents a single byte character of the format shown by the format label. Each character consumes a single byte in the data stream.

Integer

The MIG fundamental types MACH_MSG_TYPE_INTEGER_8, MACH_MSG_TYPE_INTEGER_16, MACH_MSG_TYPE_INTEGER_32 and MACH_MSG_TYPE_INTEGER_64 represent an integer of the specified length (in bits for the MIG specification). These types are encoded as a 1, 2, 4 or 8 byte value, respectively, with the endianess shown in the format label.

Floating-Point

The MIG fundamental types MACH_MSG_TYPE_REAL_32 and MACH_MSG_TYPE_REAL_64 represent single or double precision floating point values. These values are encoded as four or eight bytes, respectively, using the format shown in the format label.

Un-interpreted Byte

The MIG fundamental type MACH_MSG_TYPE_BYTE represents a single un-interpreted byte. It is encoded in the data stream as a single byte. No format conversion takes place for this value.

Port Right

When a parameter to a MIG stub is a port right (as opposed to a port name), the right must be expressed in the port right descriptors in the message body. Port rights do not consume any bytes in the data stream.

Structure Format

Since MIG can only specify homogenous structures (with each element being a fixed size), the rule for structure encoding is simple: the structure is encoded as the set of structure members packed together in their declared order within the structure with the set at an alignment boundary appropriate for any member of the structure. If, however, the data elements of the structure are port rights, no data is encoded in the data stream; the (fixed number of) port rights will be described in the port right descriptor arrays.

Array Format

MIG arrays are a singly dimensioned set of a (possibly variable) number of fixed sized entities. These entities will be encoded in the data stream packed one after the other in index order (array element 0, then 1, etc.) at a data alignment appropriate for any given member.

For fixed sized arrays, only the data values will be encoded (unless they are port rights, in which case they will be described instead in the port right descriptor arrays).

Bounded Variable-Sized Arrays

MIG variable (but bounded) arrays are encoded as NDR “conformant” arrays. Prior to the first element of the array will be a four byte current length value for the array. If the elements of the array are port rights, the port rights will be described in the port right descriptor arrays but the count field will be encoded in the data stream at the point where the array would have appeared.

CountInOut Parameter

When sending a reply message containing a variable-sized array, it is desirable to send only as much as the requestor will accept. For this purpose, MIG has the *countinout* IPC flag. From the point of view of the data stream representation, this flag causes the request message to encode the requestor’s count parameter as an additional data stream value. This extra value is not encoded in the reply data stream (just the array so bounded).

Fixed-Sized Strings

The MIG fixed-sized C string declaration is treated as the equivalent fixed-sized array of characters.

Variable-Sized Strings

The MIG variable-sized C string data is encoded using the NDR “noncorformant” string type. This representation consists of the offset of the sent string within the original string (always zero for MIG), the number of characters sent (including the terminator) followed by the characters themselves.

Co-Bounded Variable-Sized Arrays

If several (adjacent) MIG parameters are variable-sized arrays known to be of the same extent (*samecount* keyword), the first array is encoded as a bounded variable-sized array and the others are encoded as fixed-sized arrays (of the size of the first array).

Out-of-Line Format

For each “pointer” type parameter in the MIG specification, a distinct out-of-line region will be sent. The data so represented does not appear in the message body. The out-of-line region must have the alignment required by its contents. Any values necessary to describe the out-of-line region (array sizes) will be encoded into the message body portion of the data stream at the appropriate point.

Note that NDR "pointer" types (intended to transmit pointers) are not used for MIG "pointer" types (intended to specify out-of-line transmission).

APPENDIX B MIG Server Routines

This appendix describes server message de-multiplexing routines generated by MIG from the kernel interface definitions of use to a server in handling messages sent from the kernel.

Each of these routines take two parameters: a pointer to a request message and a pointer to a reply message. The request message is in the format chosen by the sender of that message (i.e., not necessarily the local machine's format); the reply message must be in the format of the local machine.

Status:

Believed to be stable

clock_reply_server

Function — Handle kernel generated clock alarms

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t clock_reply_server
        (mach_msg_header_t*          in_msg,
         mach_msg_header_t*          out_msg);
```

DESCRIPTION

The **clock_reply_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to clock alarms. Such messages are delivered to the alarm reply port named in a **clock_alarm** call. The **clock_reply_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The alarm message received from the kernel.

out_msg
: [out structure] Not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the alarm mechanism and no other action was taken.

RELATED INFORMATION

Functions: **clock_alarm_reply**.

device_reply_server

Status:

Believed to be stable

Function — Handles messages from a kernel device driver

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
boolean_t device_reply_server
        (mach_msg_header_t*        in_msg,
         mach_msg_header_t*        out_msg);
```

DESCRIPTION

The **device_reply_server** function is the MIG generated server handling function to handle messages from kernel device drivers. Such messages were sent in response to the various **device_..._request**... calls. It is assumed when using those calls that some task is listening for reply messages on the port named as a reply port to those calls. The **device_reply_server** function performs all necessary argument handling for a kernel message and calls one of the device server functions to interpret the message.

PARAMETERS

in_msg
: [pointer to in structure] The device driver message received from the kernel.

out_msg
: [out structure] A reply message. No messages from a device driver expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this device handler interface and no other action was taken.

RELATED INFORMATION

Functions: **ds_device_open_reply**, **ds_device_write_reply**, **ds_device_write_reply_inband**, **ds_device_read_reply**, **ds_device_read_reply_inband**, **ds_device_read_reply_overwrite**, **receive_packet**.

exc_server

Status:

Believed to be stable

Function — Handles kernel messages for an exception handler

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
boolean_t exc_server
        (mach_msg_header_t*                    in_msg,
         mach_msg_header_t*                   out_msg);
```

DESCRIPTION

The **exc_server** function is the MIG generated server handling function to handle messages from the kernel relating to the occurrence of an exception in a thread. Such messages are delivered to the exception port set via **thread_set_exception_port** or **task_set_exception_port**. When an exception occurs in a thread, the thread sends an exception message to its exception port, blocking in the kernel waiting for the receipt of a reply. The **exc_server** function performs all necessary argument handling for this kernel message and calls **catch_exception_raise**, **catch_exception_raise_state** or **catch_exception_raise_state_identity**, which should handle the exception. If the called routine returns KERN_SUCCESS, a reply message will be sent, allowing the thread to continue from the point of the exception; otherwise, no reply message is sent and the called routine must have dealt with the exception thread directly.

PARAMETERS

in_msg
: [pointer to in structure] The exception message received from the kernel.

out_msg
: [out structure] A reply message.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the exception mechanism and no other action was taken.

RELATED INFORMATION

Functions: **catch_exception_raise**.

memory_object_default_server

Status:

Believed to be stable

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. Note that no kernel messages to a memory manager expect a direct reply.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **seqnos_memory_object_default_server**, **memory_object_server**, **memory_object_create**, **memory_object_data_initialize**, **default_pager_object_create**, **default_pager_info**, **default_pager_paging_file**.

memory_object_server

Function — Handles kernel messages for a memory manager

Status:

Believed to be stable

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t memory_object_server
        (mach_msg_header_t*        in_msg,
         mach_msg_header_t*        out_msg);
```

DESCRIPTION

The **memory_object_server** function is the MIG generated server handling function to handle messages from the kernel targeted to a memory manager.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **memory_object_server** function performs all necessary argument handling for a kernel message and calls one of the memory manager functions to interpret the message.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **memory_object_default_server**, **memory_object_data_request**, **memory_object_data_return**, **memory_object_data_unlock**, **memory_object_notify**, **memory_object_lock_completed**, **memory_object_change_completed**, **memory_object_supply_completed**, **memory_object_terminate**, **memory_object_rejected**, **memory_object_synchronize**, **memory_object_force_copy_completed**, **seqnos_memory_object_server**.

notify_server

Status:

Believed to be stable

Function — Handle kernel generated IPC notifications

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t notify_server
        (mach_msg_header_t*        in_msg,
         mach_msg_header_t*        out_msg);
```

DESCRIPTION

The **notify_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a **mach_msg** or **mach_port_request_notification** call. The **notify_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The notification message received from the kernel.

out_msg
: [out structure] Not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the notification mechanism and no other action was taken.

RELATED INFORMATION

Functions: **seqnos_notify_server**, **do_mach_notify_dead_name**, **do_mach_-notify_no_senders**, **do_mach_notify_port_deleted**, **do_mach_notify_sen-d_once**.

prof_server

Function — Handle kernel generated PC sample messages

Status:

Believed to be stable

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t prof_server
        (mach_msg_header_t*        in_msg,
         mach_msg_header_t*        out_msg);
```

DESCRIPTION

The **prof_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to program counter (profiling) samples. Such messages are delivered to the task or thread sample port set by **task_sample** or **thread_sample**. The **prof_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The sample message received from the kernel.

out_msg
: [out structure] Not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the sample mechanism and no other action was taken.

RELATED INFORMATION

Functions: **receive_samples**.

seqnos_memory_object_default_server

Status:

Believed to be stable

Function — Handles kernel messages for the default memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t **seqnos_memory_object_default_server**
(mach_msg_header_t* *in_msg*,
mach_msg_header_t* *out_msg*);

DESCRIPTION

The **seqnos_memory_object_default_server** function is the MIG generated server handling function to handle messages from the kernel targeted to the default memory manager. This server function only handles messages unique to the default memory manager. Messages that are common to all memory managers are handled by **seqnos_memory_object_server**.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **seqnos_memory_object_default_server** function performs all necessary argument handling for a kernel message and calls one of the default memory manager functions.

PARAMETERS

in_msg
: [pointer to in structure] The memory manager message received from the kernel.

out_msg
: [out structure] A reply message. Note that no kernel messages to a memory manager expect a direct reply.

NOTES

seqnos_memory_object_default_server differs from **memory_object_default_server** in that it supplies message sequence numbers to the server interfaces it calls.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **memory_object_default_server**, **seqnos_memory_object_server**, **seqnos_memory_object_create**, **seqnos_memory_object_data_initialize**, **seqnos_default_pager_object_create**, **seqnos_default_pager_info**, **seqnos_default_pager_paging_file**.

Status:

Believed to be stable

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_data_request**, **seqnos_memory_object_data_unlock**, **seqnos_memory_object_data_return**, **seqnos_memory_object_supply_completed**, **seqnos_memory_object_notify**, **seqnos_memory_object_lock_completed**, **seqnos_seqnos_memory_object_change_completed**, **seqnos_memory_object_terminate**, **seqnos_memory_object_synchronize**, **seqnos_memory_object_rejected**, **seqnos_memory_object_force_copy_completed**, **memory_object_server**.

Status:

Believed to be stable

seqnos_notify_server

Function — Handle kernel generated IPC notifications

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

```
boolean_t seqnos_notify_server
        (mach_msg_header_t*                in_msg,
         mach_msg_header_t*                out_msg);
```

DESCRIPTION

The **seqnos_notify_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a **mach_msg** or **mach_port_request_notification** call. The **seqnos_notify_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
: [pointer to in structure] The notification message received from the kernel.

out_msg
: [out structure] Not used.

NOTES

seqnos_notify_server differs from **notify_server** in that it supplies message sequence numbers to the server interfaces.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the notification mechanism and no other action was taken.

RELATED INFORMATION

Functions: **notify_server**, **do_seqnos_mach_notify_dead_name**, **do_seqnos_-mach_notify_no_senders**, **do_seqnos_mach_notify_port_deleted**, **do_seqnos_mach_notify_send_once**.

APPENDIX C

Default Memory Management Interface

In general, the default memory manager is just like any other memory manager, except that it is “trusted” to respond promptly to paging requests in as much as that it is the memory manager of last resort. There are a few special requests issued to the default memory manager having to do with the creation and management of anonymous memory.

The following interfaces will likely be separated into a different document in a future release of the specification:

- default_pager_add_segment
- default_pager_backing_store_create
- default_pager_backing_store_delete
- default_pager_backing_store_info
- default_pager_objects
- default_pager_object_pages

Status:

New – expanded DMMI suite.
[OSF/1 R1.3]

default_pager_add_segment

Server Interface —Add additional backing storage for a default pager

LIBRARY

libmach.a only

#include <**mach/default_pager_object.h**>

SYNOPSIS

```
kern_return_t default_pager_add_segment
                (mach_port_t          backing_store,
                 mach_port_t          device,
                 recnum_t             offset,
                 recnum_t             count,
                 integer_t            record_size);
```

DESCRIPTION

The **default_pager_add_segment** function is called to add a partition to a default pager's backing storage (i.e. expand the amount of backing storage available to a memory manager). The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by tasks holding the backing store port, created with **default_pager_backing_store_create**, for a default memory manager.

The result is that the pager may use *count* records on *device* starting at *offset* for paging, and each record is *record_size* bytes in length (note that the **device_*** calls are, or can be, record oriented).

PARAMETERS

backing_store
: [in backing store (receive) right] The backing store port.

device
: [in device port] The port for the device containing the backing storage partition.

offset
: [in scalar] The offset, in *record_size* units, to the beginning of the backing storage on the device.

count
: [in scalar] The number of units ($\log_2$ of `vm_page_size`/*record_size*) in the partition/segment.

record_size
[in scalar] The size, in bytes, of the storage device record.

RETURN VALUE

KERN_FAILURE
The default pager does not support this operation.

KERN_INVALID_ARGUMENT
The *backing_store* port does not represent a valid backing store or the specified segment overlaps an existing partition.

KERN_SUCCESS
The operation was successful.

RELATED INFORMATION

Functions: **default_pager_backing_store_create**, **default_pager_backing_-store_delete**, **default_pager_backing_store_info**.

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

default_pager_backing_store_create

Server Interface — Create a backing storage object

LIBRARY

libmach.a only

#include **<mach/default_pager_object.h>**

SYNOPSIS

```
kern_return_t default_pager_backing_store_create
                (mach_port_t                pager,
                 integer_t                  priority,
                 integer_t                  clsize
                 mach_port_t*               backing_store);
```

DESCRIPTION

The **default_pager_backing_store_create** function is called to create a new backing storage object. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by tasks (privileged) holding the default pager service port.

PARAMETERS

pager
: [in default pager (receive) right] The default pager service port.

priority
: [in scalar] The scheduling priority for the backing store service thread(s).

clsize
: [in scalar] The preferred cluster size (in bytes) for the backing store object.

backing_store
: [out backing store (receive) right] The port used to manipulate the created backing store.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *pager* port does not represent a valid default pager.

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_backing_store_add_segment**, **default_pager_backing_store_delete**, **default_pager_backing_store_info**.

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

default_pager_backing_store_delete

Server Interface — Delete a backing storage object

LIBRARY

libmach.a only

#include **<mach/default_pager_object.h>**

SYNOPSIS

kern_return_t **default_pager_backing_store_delete**
(mach_port_t *backing_store*);

DESCRIPTION

The **default_pager_backing_store_delete** function is called to destroy a backing storage object created by **default_pager_backing_store_create**. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by tasks holding the backing store port, created with **default_pager_backing_store_create**, for a default memory manager.

PARAMETERS

backing_store
: [in backing store (receive) right] The backing store port created by **default_pager_backing_store_create**.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *backing_store* port does not represent a valid backing store or the specified segment overlaps an existing partition.

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_backing_store_add_segment**, **default_pager_backing_store_create**, **default_pager_backing_store_info**.

default_pager_backing_store_info

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

Server Interface — Return information about a backing storage object

LIBRARY

libmach.a only

#include **<default_pager/mach/default_pager_types.h>**

SYNOPSIS

```
kern_return_t default_pager_backing_store_info
                (mach_port_t                    backing_store
                 backing_store_flavour_t        flavor,
                 backing_store_info_t           info,
                 mach_msg_type_number_t*        size);
```

DESCRIPTION

The **default_pager_backing_store_info** function is called to obtain information about a backing storage object created by **default_pager_backing_store_create**. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by tasks holding the backing store port, created with **default_pager_backing_store_create**, for a default memory manager.

PARAMETERS

backing_store
: [in backing store (receive) right] The backing store port for which information is desired.

flavor
: [in scalar] The type of information to be changed. Valid values are:

  BACKING_STORE_BASIC_INFO
  : Statistical and space used/available information. It includes the priority and cluster size that was provided in the **default_pager_backing_store_create** call.

info
: [pointer to in structure] The data structure that will be filled-in with the information provided for the requested *flavor*.

size
: [pointer to in/out scalar] On input, the maximum size of the *info* data structure; on output, the actual size of the returned data.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *backing_store* port does not represent a valid backing store, *flavor* is not valid, or *size* is not the size for the requested flavor..

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_backing_store_add_segment**, **default_pager_backing_store_create**, **default_pager_backing_store_delete**.

Data Structures: **backing_store_basic_info**.

default_pager_info

Status:

Believed to be stable

Server Interface —Return backing storage information

LIBRARY

libmach.a only

#include **<mach/default_pager_object.h>**

SYNOPSIS

```
kern_return_t default_pager_info
                (default_pager_port_t                pager,
                 default_pager_info_t*               info);
```

seqnos_default_pager_info

Sequence Number form

```
kern_return_t seqnos_default_pager_info
                (default_pager_port_t                pager,
                 mach_port_seqno_t                   seqno,
                 default_pager_info_t*               info);
```

DESCRIPTION

A **default_pager_info** function is called as the result of a message requesting that the default memory manager return information concerning the default pager's paging partitions. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port.

PARAMETERS

pager
: [in default-pager (receive) right] The default memory manager service port.

seqno
: [in scalar] The sequence number of this message relative to the pager port.

info
: [out structure] Total and free space consumption.

RETURN VALUE

The default memory manager should return KERN_SUCCESS if it returns the desired information and KERN_FAILURE if it does not support the operation.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**, **memory_object_default_-server**, **seqnos_memory_object_default_server**.

Data Structures: **default_pager_info**.

default_pager_object_create

Status:

Believed to be stable.

Poor (but privileged) man's shared memory

Server Interface — Create a memory object managed by the default pager

LIBRARY

#include **<mach/default_pager_object.h>**

SYNOPSIS

kern_return_t **default_pager_object_create**

(default_pager_port_t	*pager,*
memory_object_t*	*memory_object,*
vm_size_t	*object_size)*;

seqnos_default_pager_object_create

Sequence Number form

kern_return_t **seqnos_default_pager_object_create**

(default_pager_port_t	*pager,*
mach_port_seqno_t	*seqno*,
memory_object_t*	*memory_object,*
vm_size_t	*object_size*);

DESCRIPTION

A **default_pager_object_create** function is called as the result of a message requesting that the default memory manager create and return a (shared) memory object which is suitable for use with **vm_map**. This memory object has the same properties as does a memory object provided by **vm_allocate**: its initial contents are zero and the backing contents are temporary in that they do not persist after the memory object is destroyed. The memory object is suitable for use as non-permanent shared memory. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port. This call should be contrasted with the kernel's **memory_object_create** message, in which the memory cache object is already created and the identity of the abstract memory object is made known to the default manager.

PARAMETERS

pager
: [in default-pager (receive) right] The default memory manager service port.

seqno
: [in scalar] The sequence number of this message relative to the *pager* port.

memory_object
: [out memory-object-representative send right] A memory object representative port (with full access) for the memory object.

object_size
: [in scalar] The maximum size for the memory object.

RETURN VALUE

Return KERN_SUCCESS if the object was created.

RELATED INFORMATION

Functions: **vm_map**, **vm_set_default_memory_manager**, **memory_object_create**, **memory_object_default_server**, **seqnos_memory_object_default_server**.

default_pager_objects

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

Server Interface — Return information about memory objects managed by a default pager

LIBRARY

libmach.a only

#include <**default_pager/mach/default_pager_types.h**>

SYNOPSIS

kern_return_t **default_pager_objects**

(mach_port_t	*pager,*
default_pager_object_array_t*	*objectsp*,
mach_msg_type_number_t*	*ocount*,
mach_port_array_t*	*portsp*,
mach_msg_type_number_t*	*pcount*);

DESCRIPTION

The **default_pager_objects** function is called to obtain information about the objects managed by a default pager. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port.

PARAMETERS

pager

[in backing store (receive) right] The backing store port for which information is desired.

objectsp

[out array of type default_pager_object_t] An array of memory objects managed by the default pager. Each element of the array has the following format:

```
struct default_pager_object {
        vm_offset_t           dpo_object;
        vm_size_t             dpo_size;
}
```

The `dpo_offset` describes the object managed by the pager; the `dpo_size` describes the backing store used for the object.

ocount
: [pointer to in/out scalar] On input, points to the size of *objectsp* in default_pager_object units; on output, indicates the number of valid entries in the array.

portsp
: [out array of mach_port_t] An array of ports for the memory objects managed by the default pager. The offset of each port is the same as the offset for the complimentary information in the *objectsp* array.

pcount
: [pointer to in/out scalar] On input, indicates the size of *portsp* in mach_port_t units; on output, indicates the number of valid entries in the array.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *pager* port does not represent a valid default pager.

KERN_RESOURCE_SHORTAGE
: A kernel memory allocation error occured while performing the operation.

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_object_pages**.

default_pager_object_pages

Status:

New – expanded DMMI suite. **[OSF/1 R1.3]**

Server Interface — Return information about memory objects managed by a default pager

LIBRARY

libmach.a only

#include <**default_pager/mach/default_pager_types.h**>

SYNOPSIS

```
kern_return_t default_pager_object_pages
        (mach_port_t                         pager,
         abs_mem_obj_port_t                  memory_object,
         default_pager_page_array_t*         pages
         integer_t*                          count);
```

DESCRIPTION

The **default_pager_object_pages** function is called to obtain information about the objects managed by a default pager. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port.

PARAMETERS

pager
: [in default-pager receive right] The default pager which manages the memory object for which information is desired.

memory_object
: [in abstract-memory-object send right] The memory object port to be examined. The port is the same as the ports returned by the **default_pager_objects** call.

pages
: [out array of type default_pager_page_t] The array of page offsets comprising the memory object specified. Each element of the array has the following format:

```
struct default_pager_page {
        vm_offset_t                dpp_offset;
}
```

The `dpp_offset` describes the offset of the page in its object.

count
: [out scalar] The number of entries in the *pages* array.

RETURN VALUE

KERN_FAILURE
: The default pager does not support this operation.

KERN_INVALID_ARGUMENT
: The *pager* port does not represent a valid default pager or the specified memory object was not found.

KERN_SUCCESS
: The operation was successful.

RELATED INFORMATION

Functions: **default_pager_objects**.

default_pager_paging_file

Status:

Obsolete – To Be Deleted
[OSF/1 R1.3]

Server Interface — Configure temporary backing storage

LIBRARY

libmach.a only

#include **<mach/default_pager_object.h>**

SYNOPSIS

```
kern_return_t default_pager_paging_file
                (default_pager_port_t          pager,
                 default_pager_filename_t      file_name,
                 boolean_t                     add);
```

seqnos_default_pager_paging_file

Sequence Number form

```
kern_return_t seqnos_default_pager_paging_file
                (default_pager_port_t          pager,
                 mach_port_seqno_t             seqno,
                 default_pager_filename_t      file_name,
                 boolean_t                     add);
```

DESCRIPTION

A **default_pager_paging_file** function is called as the result of a message requesting that the default memory manager change its backing storage configuration. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding the default memory manager port.

PARAMETERS

pager
: [in default-pager (receive) right] The default memory manager service port.

seqno
: [in scalar] The sequence number of this message relative to the pager port.

file_name
: [pointer to in array of *char*] A name of a paging partition or file.

add

[in scalar] True if the specified paging space should be added to the manager's set of space; false if it should be removed.

RETURN VALUE

The default memory manager should return KERN_SUCCESS if it performs the desired change and KERN_FAILURE if it does not support the operation.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**, **memory_object_default_-server**, **seqnos_memory_object_default_server**.

host_set_default_memory_manager

Status:

New –
replaces vm_set_default_memory_manager.

Function — Sets the default memory manager.

SYNOPSIS

```
kern_return_t host_set_default_memory_manager
                (host_control_port_t        host_priv,
                 default_pager_port_t*      default_manager,
                 vm_size_t                  cluster_size);
```

DESCRIPTION

The **host_set_default_memory_manager** function establishes the default memory manager for a host. The named manager will be the target for future **memory_object_create** calls.

PARAMETERS

host_priv
: [in host-control send right] The control port naming the host for which the default memory manager is to be set.

default_manager
: [pointer to in/out default-pager send right] A memory manager port to the new default memory manager. If this value is MACH_PORT_NULL, the old memory manager is not changed. The old memory manager port is returned in this variable.

cluster_size
: [in scalar] The preferred cluster size (in bytes) for temporary memory objects.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **memory_object_create**, **vm_allocate**.

Status:

Believed to be stable

memory_object_create

Server Interface — Requests transfer of responsibility for a kernel-created memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t memory_object_create
        (default_pager_port_t          pager,
         abs_mem_obj_port_t            new_memory_object,
         vm_size_t                     new_object_size,
         mem_cache_control_port_t      new_control,
         vm_size_t                     new_page_size);
```

seqnos_memory_object_create

Sequence Number form

```
kern_return_t seqnos_memory_object_create
        (default_pager_port_t          pager,
         mach_port_seqno_t             seqno,
         abs_mem_obj_port_t            new_memory_object,
         vm_size_t                     new_object_size,
         mem_cache_control_port_t      new_control,
         vm_size_t                     new_page_size);
```

DESCRIPTION

A **memory_object_create** function is called as the result of a message from the kernel requesting that the default memory manager accept responsibility for the new memory object created by the kernel. The kernel makes this call only to the system default memory manager.

The new memory object initially consists of zero-filled pages. Only memory pages that are actually written are provided to the memory manager. When processing **memory_object_data_request** calls from the kernel, the default memory manager must use **memory_object_data_unavailable** for any pages that have not been written previously.

The kernel does not expect a reply to this call. The kernel assumes that the default memory manager will be ready to handle data requests to this object and does not need the confirmation of a **memory_object_establish** call.

PARAMETERS

pager
: [in default-pager (receive) right] The default memory manager service port.

seqno
: [in scalar] The sequence number of this message relative to the pager port.

new_memory_object
: [in abstract-memory-object receive right] The port representing the new abstract memory object created by the kernel. (Since there will be no **memory_object_notify / memory_object_establish** sequence, no memory object representative is created.)

new_object_size
: [in scalar] The expected size for the new object, in bytes.

new_control
: [in memory-cache-control send right] The memory cache control port to be used by the memory manager when making cache management requests for the new object.

new_page_size
: [in scalar] The page size used by the kernel. All calls involving this kernel must use data sizes that are integral multiples of this page size.

NOTES

The kernel requires memory objects to provide temporary backing storage for zero-filled memory created by **vm_allocate** calls, issued by both user tasks and the kernel itself. The kernel allocates an abstract memory object port to represent the temporary backing storage and uses **memory_object_create** to pass the new memory object to the default memory manager, which provides the storage.

The default memory manager is a trusted system component that is identified to the kernel at system initialization time. The default memory manager can also be changed at run time using the **vm_set_default_memory_manager** call.

The contents of a kernel-created (as opposed to a user-created) memory object can be modified only in main memory. The default memory manager must not change the contents of a temporary memory object, or allow unrelated tasks to access the memory object, control, or name port.

The kernel provides the size of a temporary memory object based on the allocated size. Since the object is not mapped by other tasks, the object will not grow by explicit action. However, the kernel may coalesce adjacent temporary objects in such a way that this object may appear to grow. As such, the supplied object size is merely a hint as to the maximum size.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY causes **mach_msg_server** to remove the abstract memory object and memory cache control port references.

RELATED INFORMATION

Functions: **default_pager_object_create**, **memory_object_data_initialize**, **memory_object_data_unavailable**, **memory_object_default_server**, **seqnos_memory_object_default_server**.

memory_object_data_initialize

Status:

Believed to be stable

Server Interface — Writes initial data back to a temporary memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **memory_object_data_initialize**

(abs_mem_obj_port_t	*memory_object,*
mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
pointer_t	*data,*
vm_size_t	*data_count*);

seqnos_memory_object_data_initialize

Sequence Number form

kern_return_t **seqnos_memory_object_data_initialize**

(abs_mem_obj_port_t	*memory_object,*
mach_port_seqno_t	*seqno*,
mem_cache_control_port_t	*memory_control,*
vm_offset_t	*offset,*
pointer_t	*data,*
vm_size_t	*data_count*);

DESCRIPTION

A **memory_object_data_initialize** function is called as the result of a kernel message providing the default memory manager with initial data for a kernel-created memory object. If the memory manager already has supplied data (by a previous **memory_object_data_initialize** or **memory_object_data_return**), it should ignore this call. Otherwise, the call behaves the same as the **memory_object_data_return** call.

The kernel makes this call only to the default memory manager and only on temporary memory objects that it has created with **memory_object_create**.

PARAMETERS

memory_object

[in abstract-memory-object (receive) right] The abstract memory object port that represents the memory object data, as supplied by the kernel in a **memory_object_create** call.

seqno

[in scalar] The sequence number of this message relative to the abstract memory object port.

memory_control

[in memory-cache-control send right] The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset

[in scalar] The offset within the memory object.

data

[in pointer to dynamic array of bytes] The data that has been modified while cached in physical memory.

data_count

[in scalar] The number of bytes to be written, starting at *offset*. The number converts to an integral number of memory object pages.

RETURN VALUE

Any return value other than KERN_SUCCESS or MIG_NO_REPLY causes **mach_msg_server** to remove the memory cache control port reference and to de-allocate the returned data.

RELATED INFORMATION

Functions: **memory_object_create**, **memory_object_data_return**, **memory_object_default_server**, **seqnos_memory_object_default_server**.

vm_set_default_memory_manager

Status:

Obsolete – replaced by host_set_default_memory_manager.

Function — Sets the default memory manager.

SYNOPSIS

kern_return_t **vm_set_default_memory_manager**

(host_control_port_t	*host_priv,*
default_pager_port_t*	*default_manager*,
vm_size_t	*cluster_size*);

DESCRIPTION

The **vm_set_default_memory_manager** function establishes the default memory manager for a host. The named manager will be the target for future **memory_object_create** calls.

PARAMETERS

host_priv
: [in host-control send right] The control port naming the host for which the default memory manager is to be set.

default_manager
: [pointer to in/out default-pager send right] A memory manager port to the new default memory manager. If this value is MACH_PORT_NULL, the old memory manager is not changed. The old memory manager port is returned in this variable.

cluster_size
: [in scalar] The preferred cluster size (in bytes) for temporary memory objects.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **memory_object_create**, **vm_allocate**.

APPENDIX D 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. This support transparently distributes Mach IPC and virtual memory. However, each host (called a *node*) within the multicomputer maintains its identity (separate control and name ports, processor sets, devices, etc.).

This appendix describes operations that apply to individual nodes in such a configuration.

norma_get_special_port

Status:

Believed to be stable

Function — Returns a send right to a node specific port

LIBRARY

#include **<mach/norma_special_ports.h>**

SYNOPSIS

```
kern_return_t norma_get_special_port
                (host_control_port_t                host_priv,
                node_t                                   node,
                norma_special_port_t               which_port,
                mach_port_t*                     special_port);
```

norma_get_device_port

Macro form

```
kern_return_t norma_get_device_port
                (host_control_port_t                host_priv,
                node_t                                   node,
                mach_port_t*                     special_port)
```

⇒ **norma_get_special_port** (*host_priv*, *node*, NORMA_DEVICE_PORT, *special_port*)

norma_get_host_paging_port

Macro form

```
kern_return_t norma_get_host_paging_port
                (host_control_port_t                host_priv,
                node_t                                   node,
                mach_port_t*                     special_port)
```

⇒ **norma_get_special_port** (*host_priv*, *node*, NORMA_HOST_PAGING_PORT, *special_port*)

norma_get_host_port

Macro form

```
kern_return_t norma_get_host_port
                (host_control_port_t                host_priv,
                node_t                                   node,
                mach_port_t*                     special_port)
```

⇒ **norma_get_special_port** (*host_priv*, *node*, NORMA_HOST_PORT, *special_port*)

norma_get_host_priv_port

Macro form

```
kern_return_t norma_get_host_priv_port
                (host_control_port_t                host_priv,
```

node_t *node*,
mach_port_t* *special_port)*

⇒ **norma_get_special_port** (*host_priv*, *node*, NORMA_HOST_PRIV_PORT, *special_port*)

norma_get_security_port

Macro form

kern_return_t **norma_get_security_port**
(host_control_port_t *host_priv,*
node_t *node*,
mach_port_t* *special_port)*

⇒ **norma_get_special_port** (*host_priv*, *node*, NORMA_SECURITY_PORT, *special_port*)

norma_get_nameserver_port

Macro form

kern_return_t **norma_get_nameserver_port**
(host_control_port_t *host_priv,*
node_t *node*,
mach_port_t* *special_port)*

⇒ **norma_get_special_port** (*host_priv*, *node*, NORMA_NAMESERVER_PORT, *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, host control, and security ports 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.

PARAMETERS

host_priv
[in host-control send right] 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
: [device-master send right] The device master port for the node.

NORMA_HOST_PAGING_PORT
: [default-pager send right] The default pager port for the node.

NORMA_HOST_PORT
: [host-name send right] 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 default) by **mach_host_self**.

NORMA_HOST_PRIV_PORT
: [host-control send right] The host control port for the node.

NORMA_SECURITY_PORT
: [security send right] The security port for the node.

NORMA_NAMESERVER_PORT
: [name-server send right] The registered name server port for the node.

special_port
: [out norma-special send right] The returned value for the port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_host_self**, **norma_set_special_port**, **vm_set_default_memory_manager**, **bootstrap_ports**.

norma_port_location_hint

Status:

Believed to be stable

Function — Guess a port's current location

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t norma_port_location_hint
                (task_port_t                      task,
                 mach_port_name_t                 name,
                 node_t*                          node);
```

DESCRIPTION

The **norma_port_location_hint** function returns the best guess of *name*'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 task send right] Task containing the right to locate

name
: [in scalar] Name of the right to locate

node
: [out scalar] Port location hint

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **norma_task_create**.

Status:

Believed to be stable [OSF-RI NMK17].

norma_node_self

Function — Return the node index of the current host

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t norma_node_self
                (host_t                  host,
                 node_t *                node);
```

DESCRIPTION

The **norma_node_self** function returns the node index of the current host.

PARAMETERS

host
: [in host send-right] Name of the host.

node
: [out scalar] Node index of the host.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **norma_task_create**, **norma_task_clone**.

norma_set_special_port

Status:

Believed to be stable

Function — Sets a node specific special port

LIBRARY

#include **<mach/norma_special_ports.h>**

SYNOPSIS

kern_return_t **norma_set_special_port**
    (host_control_port_t *host_priv,*
    node_t *node,*
    norma_special_port_t *which_port,*
    mach_port_t *special_port)*;

norma_set_device_port

Macro form

kern_return_t **norma_set_device_port**
    (host_control_port_t *host_priv,*
    node_t *node*,
    mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*, NORMA_DEVICE_PORT, *special_port*)

norma_set_host_paging_port

Macro form

kern_return_t **norma_set_host_paging_port**
    (host_control_port_t *host_priv,*
    node_t *node*,
    mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*, NORMA_HOST_PAGING_PORT, *special_port*)

norma_set_host_port

Macro form

kern_return_t **norma_set_host_port**
    (host_control_port_t *host_priv,*
    node_t *node*,
    mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*, NORMA_HOST_PORT, *special_port*)

norma_set_host_priv_port

Macro form

kern_return_t **norma_set_host_priv_port**
    (host_control_port_t *host_priv,*

node_t *node*,
mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*, NORMA_HOST_PRIV_PORT, *special_port*)

norma_set_security_port

Macro form

kern_return_t **norma_set_security_port**
(host_control_port_t *host_priv,*
node_t *node*,
mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*, NORMA_SECURITY_PORT, *special_port*)

norma_set_nameserver_port

Macro form

kern_return_t **norma_set_nameserver_port**
(host_control_port_t *host_priv,*
node_t *node*,
mach_port_t *special_port)*

⇒ **norma_set_special_port** (*host_priv*, *node*, NORMA_NAMESERVER_PORT, *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, host control, and security ports 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.

PARAMETERS

host_priv
: [in host-control send right] 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
: [device-master send right] The device master port for the node.

NORMA_HOST_PAGING_PORT
: [default-pager send right] The default pager port for the node.

NORMA_HOST_PORT
: [host-name send right] The host name port for the node.

NORMA_HOST_PRIV_PORT
: [host-control send right] The host control port for the node.

NORMA_SECURITY_PORT
: [security send right] The security port for the node.

NORMA_NAMESERVER_PORT
: [name-server send right] The registered name server port for the node.

special_port
: [in norma-special send right] Send right to the new special port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_host_self**, **norma_get_special_port**, **vm_set_default_memory_manager**.

Status:

Believed to be stable

norma_task_clone

Function — "Clone" a task on a specified node

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **norma_task_clone**

(task_port_t	*parent_task,*
boolean_t	*inherit_memory*,
node_t	*child_node*,
task_port_t*	*child_task)*;

DESCRIPTION

The **norma_task_clone** function "clones" 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 parts of the parent's address space (see **vm_inherit**) regardless of the inheritance set for the parent's memory regions, although the inheritance for the child's regions will be set to that of the parent's regions. The child task initially contains no threads.

By way of comparison, tasks created by the standard **task_create** primitive are created on the same node as the parent.

Other than being created on a different node, the new task has the same properties as if created by **task_create**.

PARAMETERS

parent_task
: [in task send right] The port for the task from which to draw the child task's port rights, 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 task send right] The kernel-assigned port name for the new task.

NOTES

This call differs from **norma_task_create** in that the inheritance set for the parent's memory regions is ignored; the child always shares memory with the parent.

This call is intended to support process migration, where the inheritance semantics of **norma_task_create** would break migrated programs that depended upon sharing relationships remaining after migration.

This call is not a true task migration call, in that it does not migrate the port space, threads, and other non-address-space attributes of the task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **norma_task_create**.

Status:

Believed to be stable

norma_task_create

Function — Create a task on a specified node

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t norma_task_create
                (task_port_t            parent_task,
                 boolean_t              inherit_memory,
                 node_t                 child_node,
                 task_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.

By way of comparison, tasks created by the standard **task_create** primitive are created on the same node as the parent.

Other than being created on a different node, the new task has the same properties as if created by **task_create**.

PARAMETERS

parent_task
: [in task send right] The port for the task from which to draw the child task's port rights, 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 task send right] The kernel-assigned port name for the new task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **norma_task_clone**.

norma_task_teleport

Status:

To be modified due to new NoR-MA IPC system [OSF-RI NMK17].

Function — "Clone" a task on a specified node

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t **norma_task_teleport**

(task_port_t	*parent_task,*
boolean_t	*inherit_memory*,
node_t	*child_node*,
task_port_t*	*child_task)*;

DESCRIPTION

The **norma_task_clone** function "clones" 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 parts of the parent's address space (see **vm_inherit**) regardless of the inheritance set for the parent's memory regions, although the inheritance for the child's regions will be set to that of the parent's regions. The child task initially contains no threads. The *parent_task* is then terminated.

By way of comparison, tasks created by the standard **task_create** primitive are created on the same node as the parent.

Other than being created on a different node, the new task has the same properties as if created by **task_create**.

PARAMETERS

parent_task
: [in task send right] The port for the task from which to draw the child task's port rights, 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 task send right] The kernel-assigned port name for the new task.

NOTES

This call differs from **norma_task_clone** in that the parent task is terminated as part of the **teleport** call.

This call differs from **norma_task_create** in that the inheritance set for the parent's memory regions is ignored; the child always shares memory with the parent.

This call is intended to support process migration, where the inheritance semantics of **norma_task_create** would break migrated programs that depended upon sharing relationships remaining after migration.

This call is not a true task migration call, in that it does not migrate the port space, threads, and other non-address-space attributes of the task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **norma_task_clone**, **task_create**, **norma_task_create**.

Status:

Obsolete; to be deleted [OSF-RI NMK17].

task_set_child_node

Function — Set the node upon which future child tasks will be created

LIBRARY

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 task port] 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

Only generic errors apply.

RELATED INFORMATION

Functions: **norma_task_create**, **norma_task_clone**.

APPENDIX E

Bootstrap Server Interfaces

This chapter discusses the specifics of the functions used to support a bootstrap server The control port for a host is provided to the bootstrap task for that host. 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.

NOTE that this entire chapter may be moved to a separate document in a future revision of this specification.

Status:

Believed to be stable

bootstrap_arguments

Function — Return the arguments to the bootstrap task

SYNOPSIS

kern_return_t **bootstrap_ports**
(bootstrap_port_t *bootstrap,*
task_port_t *task,*
vm_offset_t* *arguments*,
mach_msg_type_number_t* *argumentCnt*);

DESCRIPTION

The **bootstrap_arguments** function returns a the set of arguments specified to the boot loader. The bootstrap task can act as a server on this interface for the tasks that it creates in order to pass arguments to them. This call only works against the bootstrap port given to the system bootstrap task.

The libsa_mach.a standalone Mach C runtime startup code uses this function, and **bootstrap_environment**, to initialize `argc`, `argv`, and `envp` for the entry point of tasks it creates (i.e. `main`).

PARAMETERS

bootstrap
: [in bootstrap send right] The bootstrap port of the task, obtained from **task_get_special_ports**.

task
: [in task send right] The task port for the task whose argument strings are requested.

arguments
: [out array] The byte array containing the argument strings. This is an array of *argumentCnt* bytes, containing NULL characters separating the strings.

argumentCnt
: [inout count] The number of bytes in the argument array.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **bootstrap_environment**, **bootstrap_ports**, **task_get_special_ports**.

bootstrap_environment

Status:

Believed to be stable

Function — Return the environment of the bootstrap task

SYNOPSIS

kern_return_t **bootstrap_ports**

(bootstrap_port_t	*bootstrap,*
task_port_t	*task,*
vm_offset_t*	*environment*,
mach_msg_type_number_t*	*environmentCnt*);

DESCRIPTION

The **bootstrap_environment** function returns a the set of arguments specified to the boot loader. The bootstrap task can act as a server on this interface for the tasks that it creates in order to pass arguments to them. This call only works against the bootstrap port given to the system bootstrap task.

The libsa_mach.a standalone Mach C runtime startup code uses this function, and **bootstrap_arguemnts**, to initialize `argc`, `argv`, and `envp` for the entry point of tasks it creates (i.e. `main`).

PARAMETERS

bootstrap
: [in bootstrap send right] The bootstrap port of the task, obtained from **task_get_special_ports**.

task
: [in task send right] The task port for the task whose environment is requested.

environment
: [out array] The byte array containing the environment strings. This is an array of *environmentCnt* bytes, containing NULL characters separating the strings.

environmentCnt
: [inout count] The number of bytes in the environment array.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **bootstrap_arguments**, **bootstrap_ports**, **task_get_special_ports**.

Status:

Believed to be stable

bootstrap_ports

Function — Returns the control ports for task startup

SYNOPSIS

```
kern_return_t bootstrap_ports
        (bootstrap_port_t            bootstrap,
         host_control_port_t*        host_control,
         device_master_port_t*       device_master,
         security_port_t*            security);
```

DESCRIPTION

The **bootstrap_ports** function returns a send right to the host control, host security and device master ports. This call only works against the bootstrap port (TASK_BOOTSTRAP_PORT) given to the system bootstrap task.

It is the responsibility of the bootstrap task to manage the distribution of these rights to other servers. An OS personality can serve as a server on the TASK_-BOOTSTRAP_PORT for tasks or servers that it manages, and can regulate or interpose on the ports in any way it deems necessary.

PARAMETERS

bootstrap
: [in bootstrap send right] The bootstrap port of the bootstrap task.

host_priv
: [out host-control send right] The control port for the host.

device_master
: [out device-master send right] The device master port.

security
: [out security send right] The host security port, used for setting task identity.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **device_open**, **host_get_clock_control**, **host_get_clock_service**, **host_processor_set_priv**, **host_processors**, **task_set_security_id**.

APPENDIX F Intel 386 Support

This appendix describes special kernel interfaces to support the special hardware features of the Intel 386 processor and its successors.

Status:

Believed to be stable

Aside from the special functions listed here, the Intel 386 support includes the following 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.

The following sections also contain special state flavors pertinent to special kernel functionality.

Debug State

The 386 debug registers (0 to 3) may be set or read using the i386_DEBUG_STATE flavor. The data for this flavor is an array of structures of the following format:

```
struct i386_debug_state
{
        unsigned int                    address;
        unsigned int                    size;
        unsigned int                    type;
};
#define i386_DEBUG_STATE_COUNT
            (sizeof (struct i386_debug_state)/sizeof(unsigned int))
```

To set or read multiple debug registers (starting with register 0), the caller supplies a data area that is a multiple of i386_DEBUG_STATE_COUNT in size.

Coprocessor Emulation

Software emulation of the (floating point) coprocessor can be selected with the i386_COPROC_EMUL flavor. The flavor takes a value which is a single boolean. When set TRUE, all floating point instructions will raise an exception of type EXC_EMULATION (with an exception code of EXC_I386_NOEXT).

I/O Support

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.

In the thread state→*efl* flags is a simulated interrupt request flag, EFL_IRQ. This flag can be manipulated by manipulating the i386_THREAD_STATE. As an optimization, the i386_INTERRUPT_REQUEST state flavor manipulates only this one flag, taking a boolean value as the flavor's data.

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 state 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 (EFL_IRQ), 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.

The i386_INT_BITMAP state flavor takes a bitmap of 256 bits used to control software interrupts. If bit *N* is 0 then an "INT $*N*" instruction will execute as it would in real mode; otherwise such an instruction will take a general protection exception. The default value for this bitmap is all 1 bits. This bitmap only affects virtual 8086 threads.

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. A thread may map this LDT into its address space (read-only) with the **i386_map_ldt_readonly** function.

Status:

Believed to be stable

i386_get_ldt

Function — Return per-thread segment descriptors

LIBRARY

#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
        (thread_port_t                  thread,
         int                            first_selector,
         int                            desired_count,
         descriptor_list_t*             desc_list,
         mach_msg_type_number_t*        desc_list_count);
```

DESCRIPTION

The **i386_get_ldt** function returns per-thread segment descriptors from the thread's local descriptor table (LDT).

PARAMETERS

thread
: [in thread send right] 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
: [out pointer to dynamic array of *descriptor_t*] Array of segment descriptors.

desc_list_count
: [in/out scalar] On input, the maximum size of the buffer; on output, the size returned (in natural-sized units).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **i386_set_ldt**, **i386_map_ldt_readonly**.

Status:

Believed to be stable

i386_io_port_add

Function — Permit IO instructions to be performed against a device

LIBRARY

#include **<mach/i386/mach_i386.h>**

SYNOPSIS

```
kern_return_t i386_io_port_add
                (thread_port_t                    thread,
                 device_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 thread send right] Thread whose permission bitmap is to be set.

device
: [in device send right] 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

Only generic errors apply.

RELATED INFORMATION

Functions: **i386_io_port_list**, **i386_io_port_remove**.

i386_io_port_list

Status:

Believed to be stable

Function — List devices permitting IO

LIBRARY

#include **<mach/i386/mach_i386.h>**

SYNOPSIS

kern_return_t **i386_io_port_list**

(thread_port_t	*thread,*
device_port_array_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 thread send right] Thread whose permission list is to be returned

list
: [out pointer to dynamic array of device send rights] Device ports permitting IO

count
: [out scalar] The number of ports returned

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **i386_io_port_add**, **i386_io_port_remove**.

Status:

Believed to be stable

i386_io_port_remove

Function — Disable IO instructions against a device

LIBRARY

#include **<mach/i386/mach_i386.h>**

SYNOPSIS

kern_return_t **i386_io_port_remove**
(thread_port_t *thread,*
device_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 thread send right] Thread whose permission bitmap is to be cleared

device
[in device send right] Device whose permission is to be revoked

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **i386_io_port_add**, **i386_io_port_list**.

i386_map_ldt_readonly

Status: **Believed to be stable**

Function — Directly maps (read-only) a thread's LDT into the task's address space

LIBRARY

#include **<mach/i386/mach_i386.h>**

SYNOPSIS

```
kern_return_t i386_map_ldt_readonly
                (thread_port_t                    thread,
                 vm_address_t                     address,
                 vm_size_t                        size);
```

DESCRIPTION

The **i386_map_ldt_readonly** function maps the local descriptor table of the target thread into the target threads address space, in the region designated by *address* and *size*. The mapping is read-only: changes made by **i386_set_ldt** or by the hardware (while the thread is running) are visible immediately but the region is not writable.

PARAMETERS

thread
: [in thread send right] Thread whose LDT is to be mapped

address
: [in scalar] The address of the region where the LDT is to be mapped. This must correspond to a page address.

size
: [in scalar] The size (in bytes) of the region into which the LDT is to be mapped.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **i386_get_ldt**, **i386_set_ldt**.

Status:

Believed to be stable

i386_set_ldt

Function — Set per-thread segment descriptors

LIBRARY

#include <**mach/i386/mach_i386.h**>

SYNOPSIS

```
struct descriptor
{
        unsigned int                    low_word;
        unsigned int                    high_word;
};
typedef struct descriptor               descriptor_t;
typedef struct descriptor*              descriptor_list_t;
```

```
kern_return_t i386_set_ldt
        (thread_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 thread send right] 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

Only generic errors apply.

RELATED INFORMATION

Functions: **i386_get_ldt**, **i386_map_ldt_readonly**.

APPENDIX G 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.

Status:

New – added in support of expanded DMMI suite. **[OSF/1 R1.3]**

backing_store_basic_info

Structure — Defines information concerning backing storage

SYNOPSIS

```
struct backing_store_basic_info
{
        integer_t               pageout_calls;
        integer_t               pagein_calls;
        integer_t               pages_in;
        integer_t               pages_out;
        integer_t               pages_unavail;
        integer_t               pages_init;
        integer_t               pages_init_writes;
        integer_t               bs_pages_total;
        integer_t               bs_pages_free;
        integer_t               bs_pages_in;
        integer_t               bs_pages_in_fail;
        integer_t               bs_pages_out;
        integer_t               bs_pages_out_fail;
        integer_t               bs_priority;
        integer_t               bs_clsize;
} backing_store_basic_info_t;
```

DESCRIPTION

The **backing_store_basic_info** structure defines information concerning the default memory manager's backing storage.

FIELDS

pageout_calls
: Total number of pageout calls made.

pagein_calls
: Total number of pagein calls made.

pages_in
: Total number of pages that have been paged in.

pages_out
: Total number of pages that have been paged out.

pages_unavail
: Total number of zero-fill pages.

pages_init
: Total number of page init requests.

pages_init_writes
: Total number of page init writes.

bs_pages_total
: The total number of pages on the specified backing store.

bs_pages_free
: The total number of unallocated pages on the specified backing store.

bs_pages_in
: The number of page read requests handled by the specified backing store.

bs_pages_in_fail
: The number of page read requests handled by the specified backing store that have failed.

bs_pages_out
: The number of page write requests handled by the specified backing store.

bs_pages_out_fail
: The number of page write requests handled by the specified backing store that have failed.

bs_priority
: The scheduling priority for the backing store thread(s). This value is supplied in the **default_pager_backing_store_create** call.

bs_clsize
: The backing store cluster size, in bytes. This value is supplied in the **default_pager_backing_store_create** call.

NOTES

This structure is machine word length specific because of the memory size returned.

RELATED INFORMATION

Functions: **default_pager_backing_store_info**, **default_pager_backing_store_create.**

Status:

Believed to be stable

default_pager_info

Structure — Defines information concerning temporary backing store

SYNOPSIS

```
[1] struct default_pager_info
[2] {
[3]         vm_size_t                  dpi_total_space;
[4]         vm_size_t                  dpi_free_space;
[5]         vm_size_t                  dpi_page_size;
[6] } default_pager_info_t;
```

DESCRIPTION

The **default_pager_info** structure defines information concerning the default memory manager's backing storage consumption.

FIELDS

dpi_total_space
Total space, in bytes, allocated for default memory backing storage

dpi_free_space
Free space, in bytes, in the default manager's currently allocated paging partitions

dpi_page_size
Default manager's VM page size

NOTES

This structure is machine word length specific because of the memory size returned.

RELATED INFORMATION

Functions: **default_pager_info.**

host_basic_info

Status:

Believed to be stable

Structure — Defines basic information about a host

SYNOPSIS

```
[1] struct host_basic_info
[2] {
[3]         cpu_number_t            max_cpus;
[4]         cpu_number_t            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_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

NOTES

This structure is machine word length specific because of the memory size returned.

RELATED INFORMATION

Functions: **host_info.**

Data structures: **host_load_info**, **host_sched_info.**

Status:

Believed to be stable

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]         load_t                  avenrun[3];
[7]         load_t                  mach_factor[3];
[8] };
[9] 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_statistics.**

Data structures: **host_basic_info**, **host_sched_info.**

host_sched_info

Status:

Believed to be stable

Structure — Defines scheduling information about a host

SYNOPSIS

```
[1]  struct host_sched_info
[2]  {
[3]          mach_msg_timeout_t          min_timeout;
[4]          mach_msg_timeout_t          min_quantum;
[5]  };
[6]  typedef struct host_sched_info*     host_sched_info_t;
```

DESCRIPTION

The **host_sched_info** structure defines the limiting scheduling information available about a host.

FIELDS

min_timeout
: Minimum time-out, in milliseconds

min_quantum
: Minimum quantum (period for which a thread can be scheduled if uninterrupted), in milliseconds

RELATED INFORMATION

Functions: **host_info.**

Data structures: **host_basic_info**, **host_load_info.**

Status:

New: [OSF-RI MK6] Added for asynchronous I/O completion status

io_done_result

Structure — Defines asynchronous I/O completion status

SYNOPSIS

```
[1]  struct io_done_result
[2]  {
[3]  struct control_info
[4]  {
[5]          int                      qd_type;
[6]          mach_port_t              qd_reqid;
[7]          kern_return_t            qd_code;
[8]          io_buf_ptr_t             qd_data;
[9]          int                      qd_count;
[10] } qd_control;
[11] io_buf_ptr_inband_t              qd_inline;
[12] };
[13] typedef struct io_done_result    io_done_result_t;
```

DESCRIPTION

The **io_done_result** structure defines the set of information made available by a device driver upon completion of an outstanding asynchronous I/O request.

FIELDS

qd_type
: Identifies the type of I/O operation that was requested. Valid values are:

IO_DONE_READ The operation was a device_read_async or device_-read_async_inband.

IO_DONE_OVERWRITE The operation was a device_read_over-write_async.

IO_DONE_WRITE The operation was a device_write_async or de-vice_write_async_inband.

qd_reqid
: An opaque data item used to uniquely identify the I/O request to the task. It is typically supplied by the task as an argument to the asychronous device request. The µKernel transparently returns this data item to the task.

qd_code
: The return code that would have been returned by the non-asynchronous variant of the I/O call (i.e. device_read for device_read_async). If

the field is set to KERN_SUCCESS, then the *qd_data* and *qd_count* fields are valid.

qd_data
If the qd_type field is set to IO_DONE_READ or IO_DONE_OVERWRITE, then this field is a pointer to the data; otherwise, this field's contents is undefined.

qd_count
The number of bytes read or written.

qd_inline
Contains the actual data, pointed to by *qd_data*, used in **device_read_async_inband**. In all other cases, this field is undefined.

RELATED INFORMATION

Functions: **io_done_queue_create**, **io_done_queue_wait**, **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**.

kernel_resource_sizes

Status: **Experimental**

Structure — Defines the sizes of significant kernel structures

SYNOPSIS

```
[1] struct kernel_resource_sizes
[2] {
[3]         ledger_item_t          task;
[4]         ledger_item_t          thread;
[5]         ledger_item_t          port;
[6]         ledger_item_t          memory_region;
[7]         ledger_item_t          memory_object;
[8] };
[9] typedef struct kernel_resource_sizes*    kernel_resource_sizes_t;
```

DESCRIPTION

The **kernel_resource_sizes** structure defines the sizes of significant kernel structures. This information is of use when establishing ledger resource limits.

FIELDS

task
Space consumed by an empty task.

thread
Space consumed by a thread.

port
Space consumed by a port with an empty message queue.

memory_region
Space consumed by each distinct memory region (as reported by **vm_region**) in a task's address space.

memory_object
Space consumed to manage a memory object with no resident pages or copy objects.

RELATED INFORMATION

Functions: **host_info**, **ledger_create**.

mach_msg_descriptor

Structure — Defines the disposition of kernel processed message data

Status:

Believed to be stable

SYNOPSIS

```
typedef struct
{
        void*                         pad1;
        mach_msg_size_t               pad2;
        unsigned int                  pad3: 24;
        mach_msg_descriptor_type_t    type: 8;
} mach_msg_type_descriptor_t;
typedef struct
{
        mach_port_t                   name;
        mach_msg_size_t               pad1;
        unsigned int                  pad2: 16;
        mach_msg_type_name_t          disposition: 8;
        mach_msg_descriptor_type_t    type: 8;
} mach_msg_port_descriptor_t;
typedef struct
{
        void*                         address;
        mach_msg_size_t               size;
        boolean_t                     deallocate: 8;
        mach_msg_copy_options_t       copy: 8;
        unsigned int                  pad1: 8;
        mach_msg_descriptor_type_t    type: 8;
} mach_msg_ool_descriptor_t;
typedef struct
{
        void*                         address;
        mach_msg_size_t               count;
        boolean_t                     deallocate: 8;
        mach_msg_copy_options_t       copy: 8;
        mach_msg_type_name_t          disposition: 8;
        mach_msg_descriptor_type_t    type: 8;
} mach_msg_ool_ports_descriptor_t;
typedef union
{
        mach_msg_type_descriptor_t    type;
        mach_msg_port_descriptor_t    port;
        mach_msg_ool_descriptor_t     out_of_line;
        mach_msg_ool_ports_descriptor_t ool_ports;
} mach_msg_descriptor_t;
```

DESCRIPTION

A **mach_msg_descriptor** structure describes the processing to be performed for an element of kernel processed data in a Mach message.

FIELDS

name
: For single port descriptors, the name of the port whose right is being sent.

disposition
: For single port or out-of-line port array descriptors, the IPC processing to be done for the rights for the named ports.

address
: For out-of-line data or port array descriptors, the address of the out-of-line data or port (name) array.

size
: For out-of-line data descriptors, the size of the out-of-line region, in bytes.

deallocate
: For out-of-line data descriptors, true if the set of pages containing the array should be de-allocated when the message is sent.

copy
: For out-of-line descriptors, a description of the method by which the data should be copied.

count
: For out-of-line port array descriptors, the number of port names in the array.

type
: For any type of descriptor, the type of descriptor.

port
: A descriptor that describes a single port right.

out_of_line
: A descriptor that describes an out-of-line data array.

ool_ports
: A descriptor that describes an out-of-line port array.

RELATED INFORMATION

Functions: **mach_msg**.

Data structures: **mach_msg_header**.

Status:

Believed to be stable

mach_msg_header

Structure — Defines the header portion for messages

SYNOPSIS

```
typedef struct
{
        mach_msg_bits_t             msgh_bits;
        mach_msg_size_t             msgh_size;
        mach_port_t                 msgh_remote_port;
        mach_port_t                 msgh_local_port;
        mach_msg_size_t             msgh_reserved;
        mach_msg_id_t               msgh_id;
} mach_msg_header_t;
typedef struct
{
        mach_msg_size_t             msgh_descriptor_count;
} mach_msg_body_t;
typedef struct
{
        mach_msg_trailer_type_t     msgh_trailer_type;
        mach_msg_trailer_size_t     msgh_trailer_size;
} mach_msg_trailer_t;
typedef struct
{
        mach_msg_trailer_type_t     msgh_trailer_type;
        mach_msg_trailer_size_t     msgh_trailer_size;
        mach_port_seqno_t           msgh_seqno;
} mach_msg_seqno_trailer_t;
typedef struct
{
        mach_msg_trailer_type_t     msgh_trailer_type;
        mach_msg_trailer_size_t     msgh_trailer_size;
        mach_port_seqno_t           msgh_seqno;
        security_id_t               msgh_sender;
} mach_msg_security_trailer_t;
```

DESCRIPTION

The **mach_msg_header** structure defines the fixed size header of a Mach message. The header is followed by a message body containing data and port descriptors and zero or more data bytes.

If the MACH_MSGH_BITS_COMPLEX flag in the *msgh_bits* field is not set, then this is a simple message described by **mach_msg_header_t**. In this case, the header is immediately followed by un-typed data.

If the complex flag is set, then this is a “complex” message consisting of a **mach_msg_header_t** structure followed by a **mach_msg_body_t** structure containing a count followed by an array of descriptors specifying the disposition (processing) to be performed for the out-of-line memory regions and additional port rights.

Following the header (and any kernel processed descriptors), at natural alignment can be additional (un-typed) data, up to the size of the message (*msgh_size*). This extra data typically carries information used to decode the data stream and out-of-line data.

At the next natural boundary following the message data is the message trailer (**mach_msg_trailer_t**). This structure indicates the type and length of the trailer. If the length is greater than sizeof (**mach_msg_trailer_t**), additional fields follow providing kernel generated message attributes.

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 additional port rights or out-of-line memory regions.

  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

This field is ignored on send (the size to send is specified by the *send_-size* parameter to **mach_msg**); the field is set on receive to the sum of the message header and body sizes (in bytes). Note that this value may be different from the send size specified by the sender if the sender and receiver machines have differing sizes for port names, memory addresses or memory range sizes.

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_id

Not set or read by the **mach_msg** call. The conventional meaning is to convey an operation or function ID.

msgh_descriptor_count

The number of descriptors of kernel processed data (port rights and out-of-line data).

msgh_trailer_type

An identifier of the trailer version. Different values indicate not necessarily compatible trailer formats. The current (and only) trailer format is MACH_MSG_TRAILER_FORMAT_0. There is currently only one attribute defined within this trailer type: the sender's identity.

msgh_trailer_size

The length, in bytes, of the message trailer, including the trailer type and length fields.

msgh_seqno

The sequence number of this message relative to the port from which it is received.

msgh_sender

The security ID of the sender of the message.

RELATED INFORMATION

Functions: **mach_msg**.

Data structures: **mach_msg_descriptor**.

Status:

Believed to be stable, except for the size_limit **ledger item**

mach_port_limits

Structure — Defines queue and resource limits for a port

SYNOPSIS

[1]	struct **mach_port_limits**		
[2]	{		
[3]		mach_port_msgcount_t	*queue_limit*;
[4]		mach_port_oolcount_t	*memory_limit*;
[5]		mach_port_rightcount_t	*right_limit*;
[6]		ledger_item_t	*size_limit*;
[7]	};		
[8]	typedef struct mach_port_limits*		**mach_port_limits_t**;

DESCRIPTION

The **mach_port_limits** structure defines various limits governing the messages that can be sent through the port.

FIELDS

queue_limit
Number of messages allowed to be on the message queue at any given time. Attempts to queue more messages than this limit will block.

memory_limit
Limit on the number of bytes of out-of-line data that may be received in a message.

right_limit
Limit on the number of port rights (including the reply port) that may be received in a message.

size_limit
Limit on the total size (in-line data, port structures, etc.) that may be consumed by a message queued on the port.

RELATED INFORMATION

Functions: **mach_port_get_attributes**, **mach_port_set_attributes**.

mach_port_status

Status:

Believed to be stable.

Changed mps_srights to an integer and listed potential values. **[OSF-RI DIPC]**

Structure — Defines information for a port

SYNOPSIS

```
[1]  struct mach_port_status
[2]  {
[3]          mach_port_name_t          mps_pset;
[4]          mach_port_seqno_t         mps_seqno;
[5]          mach_port_mscount_t       mps_mscount;
[6]          mach_port_msgcount_t      mps_msgcount;
[7]          mach_port_rights_t        mps_sorights;
[8]          integer_t                 mps_srights;
[9]          boolean_t                 mps_nsrequest;
[10] };
[11] 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_msgcount
Number in the queue.

mps_sorights
How many send-once rights.

mps_srights
Indicates whether the port has send rights. Valid values are:

MPS_FALSE
The port does not have send rights.

MPS_TRUE
The port has send rights (with no more senders notification enabled).

MPS_UNKNOWN
The port does not permit no more senders notification indicating that it is unknown whether send rights exist.

mps_nsrequest
True if no-senders requested.

NOTES

This structure is machine word length sensitive due to the presence of the port set name.

RELATED INFORMATION

Functions: **mach_port_get_attributes.**

mapped_tvalspec

Status:

Believed to be stable

Structure — Defines format of kernel maintained time in a mapped clock

SYNOPSIS

```
struct mapped_tvalspec
{
        tvalspec_t          mtv_time;
        nsec_t              mtv_csec;
};
typedef struct mapped_tvalspec    mapped_tvalspec_t;
```

DESCRIPTION

The **mapped_tvalspec** structure defines the format of the current-time structure maintained by the kernel and visible through a mapped clock (**clock_map_time**). The data in this structure is updated at the clock's current resolution and contains the same tvalspec value that would be returned by **clock_get_time**.

FIELDS

mtv_time
: Clock time.

mtv_csec
: A field used to synchronize with the kernel's setting of the time.

NOTES

Because of the race between the referencing of the multiple fields in the clock value and the kernel's setting them, they should be referenced as follows:

```
tvalspec_t*          ts;
do
{
        ts -> tv_sec = mtime -> mtv_time.tv_sec;
        ts -> tv_nsec = mtime -> mtv_time.tv_nsec;
} while (ts -> tv_sec != mtime -> mtv_csec);
```

RELATED INFORMATION

Functions: **clock_map_time**, **clock_get_time.**

Data Structures: **tvalspec**.

Status:

Believed to be stable

memory_object_attr_info

Structure — Defines the behavioral attributes for a memory object

SYNOPSIS

```
struct memory_object_attr_info
{
        memory_object_copy_strategy_t    copy_strategy;
        vm_offset_t                      cluster_size;
        boolean_t                        may_cache_object;
        boolean_t                        temporary;
        boolean_t                        write_completions;
};
typedef struct memory_object_attr_info*  memory_object_attr_info_t;
```

DESCRIPTION

The **memory_object_attr_info** structure defines behavior and performance relevant memory object attributes.

FIELDS

copy_strategy
: How the kernel should handle copying of regions associated with the memory object. The copy strategy cannot be changed once an object is initialized. Valid values are:

MEMORY_OBJECT_COPY_NONE
: Use normal procedure when copying the memory object's data. Normally, the kernel requests each page with read access, copies the data, and then (optionally) flushes the data.

MEMORY_OBJECT_COPY_DELAY
: Use copy-on-write technique. This strategy allows the kernel to efficiently copy large amounts of data and guarantees that the memory manager will not externally modify the data. It is the most commonly used copy strategy.

cluster_size
: Preferred cluster size (in bytes) for the memory object. This helps to determine how many pages are transferred in individual data request and return messages.

may_cache_object
: Cache indicator. If true, the kernel can cache data associated with the memory object (keep the memory object active) even if no virtual memory references to it remain.

temporary

When the last mapping of the memory object is removed, the object is destroyed without returning any in-memory pages.

write_completions

If true, the memory manager must explicitly indicate when it has disposed of returned pages with **memory_object_data_return_completed**. This attribute can only be set (implicitly) by using the privileged **memory_object_establish_priv**.

NOTES

Sharing cached data among all the clients of a memory object can have a major impact on performance, especially if it can be extended across successive, as well as concurrent, uses. For example, the memory objects that represent program images can be used regularly by different programs. By retaining the data for these memory objects in cache, the number of secondary storage accesses can be reduced significantly.

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_get_attributes**, **memory_object_change_attributes**, **vm_region, memory_object_synchronize, memory_object_data_return_completed**, **vm_set_default_memory_manager**, **memory_object_establish_priv**, **vm_msync**.

Status:

Believed to be stable

policy_fifo_info

Structure — Defines first-in-first-out scheduling information

SYNOPSIS

```
struct policy_fifo_limit
{
        priority_t              max_priority;
};
struct policy_fifo_base
{
        priority_t              base_priority;
};
struct policy_fifo_info
{
        priority_t              max_priority;
        priority_t              base_priority;
        boolean_t               depressed;
        priority_t              depress_priority;
};
```

DESCRIPTION

The **policy_fifo_info** structure defines the first-in-first-out scheduling policy information.

FIFO threads have two priorities associated with them by the system:

- A maximum priority value 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 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.

FIELDS

max_priority
Maximum scheduling priority

base_priority
Scheduling priority

depressed
True if scheduling priority is depressed

depress_priority
Scheduling priority from which depressed

RELATED INFORMATION

Functions: **thread_info**, **task_info**, **processor_set_info**, **processor_set_policy_-default**, **processor_set_policy_disable**, **processor_set_policy_enable**, **processor_set_policy_limit**, **task_policy**, **thread_policy**, **thread_set_policy**.

Data Structures: **policy_rr_info**, **policy_timeshare_info**.

policy_rr_info

Status:

Believed to be stable

Structure — Defines round-robin scheduling information

SYNOPSIS

```
struct policy_rr_limit
{
        priority_t              max_priority;
};
struct policy_rr_base
{
        priority_t              base_priority;
        mach_msg_timeout_t      quantum;
};
struct policy_rr_info
{
        priority_t              max_priority;
        priority_t              base_priority;
        mach_msg_timeout_t      quantum;
        boolean_t               depressed;
        priority_t              depress_priority;
};
```

DESCRIPTION

The **policy_rr_info** structure defines the round-robin scheduling policy information.

Round-robin threads have two priorities associated with them by the system:

- A maximum priority value 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 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.

FIELDS

max_priority
Maximum scheduling priority

base_priority
Scheduling priority

quantum
Scheduling quantum (in milliseconds)

depressed
True if scheduling priority is depressed

depress_priority
Scheduling priority from which depressed

RELATED INFORMATION

Functions: **thread_info**, **task_info**, **processor_set_info**, **processor_set_policy_default**, **processor_set_policy_disable**, **processor_set_policy_enable**, **processor_set_policy_limit**, **task_policy**, **thread_policy**, **thread_set_policy**.

Data Structures: **policy_fifo_info**, **policy_timeshare_info**.

Status:

Believed to be stable

policy_timeshare_info

Structure — Defines timeshare scheduling information

SYNOPSIS

```
[1] struct policy_timeshare_limit
[2] {
[3]         priority_t              max_priority;
[4] };
[1] struct policy_timeshare_base
[2] {
[3]         priority_t              base_priority;
[4] };
[1] struct policy_timeshare_info
[2] {
[3]         priority_t              max_priority;
[4]         priority_t              base_priority;
[5]         priority_t              cur_priority;
[6]         boolean_t               depressed;
[7]         priority_t              depress_priority;
[8] };
```

DESCRIPTION

The **policy_timeshare_info** structure defines the timeshare scheduling policy information.

Timeshare threads have three priorities associated with them by the system:

- 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 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 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).

FIELDS

max_priority
Maximum scheduling priority

base_priority
Base 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**, **task_info**, **processor_set_info**, **processor_set_policy_-default**, **processor_set_policy_disable**, **processor_set_policy_enable**, **processor_set_policy_limit**, **task_policy**, **thread_policy**, **thread_set_policy**.

Data Structures: **policy_fifo_info**, **policy_rr_info**.

Status:

Believed to be stable

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]         cpu_number_t        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

Status:

Believed to be stable

Structure — Defines the basic information about a processor set.

SYNOPSIS

```
[1]  struct processor_set_basic_info
[2]  {
[3]          cpu_number_t                     processor_count;
[4]          policy_t                         default_policy;
[5]  };
[6]  typedef struct processor_set_basic_info*     processor_set_basic_info_t;
```

DESCRIPTION

The **processor_set_basic_info** structure defines the basic information available about a processor set.

FIELDS

processor_count
Number of processors in this set

default_policy
Default policy to assign to threads whose otherwise assigned policy is not enabled

RELATED INFORMATION

Functions: **processor_set_info**.

Data Structures: **processor_set_load_info**.

Status:

Believed to be stable

processor_set_load_info

Structure — Defines the scheduling statistics for a processor set.

SYNOPSIS

```
[1]  struct processor_set_load_info
[2]  {
[3]          mach_msg_type_number_t          task_count;
[4]          mach_msg_type_number_t          thread_count;
[5]          load_t                          load_average;
[6]          load_t                          mach_factor;
[7]  };
[8]  typedef struct processor_set_load_info*  processor_set_load_info_t;
```

DESCRIPTION

The **processor_set_load_info** structure defines the scheduling statistics maintained for a processor set.

FIELDS

task_count
Number of tasks currently assigned to this processor set

thread_count
Number of threads currently assigned to this processor set

load_average
Average number of runnable processes divided by number of CPUs

mach_factor
The processing resources available to a new thread—the number of CPUs divided by (1 + the number of threads)

RELATED INFORMATION

Functions: **processor_set_statistics**.

Data Structures: **processor_set_sched_info**, **processor_set_basic_info**.

task_basic_info

Status:

Believed to be stable

Structure — Defines basic information for tasks

SYNOPSIS

```
[1]  struct task_basic_info
[2]  {
[3]          suspend_t                        suspend_count;
[4]          vm_size_t                        virtual_size;
[5]          vm_size_t                        resident_size;
[6]          tvalspec_t                       user_time;
[7]          tvalspec_t                       system_time;
[8]          policy_t                         policy;
[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.

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.

policy
Default scheduling policy to apply to new threads.

NOTES

This structure is machine word length sensitive due to the presence of the virtual address sizes.

RELATED INFORMATION

Functions: **task_info**.

Data Structures: **task_thread_times_info**, **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

task_thread_times_info

Status:

Believed to be stable

Structure — Defines thread execution times information for tasks

SYNOPSIS

[1] struct **task_thread_times_info**
[2] {
[3] tvalspec_t *user_time*;
[4] tvalspec_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**.

Status:

Believed to be stable

thread_basic_info

Structure — Defines basic information for threads

SYNOPSIS

[1] struct **thread_basic_info**
[2] {
[3] tvalspec_t *user_time*;
[4] tvalspec_t *system_time*;
[5] cpu_usage_t *cpu_usage*;
[6] policy_t *policy*;
[7] run_state_t *run_state*;
[8] thread_flags_t *flags*;
[9] suspend_t *suspend_count*;
[10] time_t *sleep_time*;
[11] };
[12] 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.

policy
Scheduling policy in effect

run_state
The thread's run state. Possible values are:

TH_STATE_RUNNING
The thread is running normally.

TH_STATE_STOPPED
The thread is stopped.

TH_STATE_WAITING
The thread is waiting normally.

TH_STATE_UNINTERRUPTIBLE
: The thread is in an un-interruptible wait state.

TH_STATE_HALTED
: The thread is halted at a clean point.

flags
: Swap/idle flags for the thread. Possible values are:

TH_FLAGS_SWAPPED
: The thread is swapped out.

TH_FLAGS_IDLE
: The thread is an idle thread.

suspend_count
: The current suspend count for the thread.

sleep_time
: The number of seconds that the thread has been sleeping.

RELATED INFORMATION

Functions: **thread_info**.

Data structures: **policy_fifo_info**, **policy_rr_info**, **policy_timeshare_info**.

Status:

Believed to be stable

tvalspec

Structure — Defines format of system time values

SYNOPSIS

```
[1]  struct tvalspec
[2]  {
[3]          time_t                      tv_sec;
[4]          nsec_t                      tv_nsec;
[5]  };
[6]  typedef struct tvalspec             tvalspec_t;
```

DESCRIPTION

The **tvalspec** structure defines the format of the time structure supplied to or returned from the kernel. This definition conforms to the Posix 1003.4 *timespec* definition where the *tv_nsec* structure member is valid if ($0 \leq tv\_nsec < 10^9$) and the time period described is (($tv\_sec \times 10^9$) + tv_nsec) nanoseconds.

FIELDS

tv_sec
: Seconds.

tv_nsec
: Nanoseconds.

RELATED INFORMATION

Functions: **clock_get_time, clock_set_time**, **clock_sleep**, **clock_alarm**, **clock_-alarm_reply.**

Data Structures: **mapped_tvalspec**.

vm_region_info

Status:

Believed to be stable

Structure — Defines the attributes of a task's memory region

SYNOPSIS

```
[1]  struct vm_region_info
[2]  {
[3]          vm_prot_t          protection;
[4]          vm_prot_t          max_protection;
[5]          vm_inherit_t       inheritance;
[6]          boolean_t          shared;
[7]          boolean_t          reserved;
[8]          vm_offset_t        offset;
[9]          vm_behavior_t      behavior;
[10] };
[11] typedef struct vm_region_info*     vm_region_info_t;
```

DESCRIPTION

The **vm_region_info** structure defines the attributes of a memory region returned by **vm_region**.

FIELDS

protection
The current protection for the region.

max_protection
The maximum protection allowed for the region.

inheritance
The inheritance attribute for the region.

shared
Shared indicator. If true, the region is shared by another task. If false, the region is not shared.

reserved
If true the region is protected from random allocation.

offset
The region's offset into the memory object. The region begins at this offset.

behavior
Expected reference pattern for the memory. Valid values are:

VM_BEHAVIOR_DEFAULT
The system's default behavior.

VM_BEHAVIOR_RANDOM
No particular order expected.

VM_BEHAVIOR_SEQUENTIAL
Forward sequential order.

VM_BEHAVIOR_RSEQNTL
Reverse sequential order.

RELATED INFORMATION

Functions: **vm_region, vm_inherit**, **vm_reserve**, **vm_behavior_set**, **vm_protect**.

vm_statistics

Status:

Believed to be stable

Structure — Defines statistics for the kernel's use of virtual memory

SYNOPSIS

```
[1]  struct vm_statistics
[2]  {
[3]          vm_size_t          free_count;
[4]          vm_size_t          active_count;
[5]          vm_size_t          inactive_count;
[6]          vm_size_t          wire_count;
[7]          vm_size_t          zero_fill_count;
[8]          vm_size_t          reactivations;
[9]          vm_size_t          pageins;
[10]         vm_size_t          pageouts;
[11]         vm_size_t          faults;
[12]         vm_size_t          cow_faults;
[13]         vm_size_t          lookups;
[14]         vm_size_t          hits;
[15] };
[16] typedef struct vm_statistics*  vm_statistics_t;
```

DESCRIPTION

The **vm_statistics** structure defines the statistics available on the kernel's use of virtual memory. The statistics record virtual memory usage since the kernel was booted.

For related information for a specific task, see the **task_basic_info** structure.

FIELDS

free_count
: The total number of free pages in the system.

active_count
: The total number of pages currently in use and pageable.

inactive_count
: The number of inactive pages.

wire_count
: The number of pages that are wired in memory and cannot be paged out.

zero_fill_count
: The number of zero-fill pages.

reactivations
The number of reactivated pages.

pageins
The number of requests for pages from a pager (such as the i-node pager).

pageouts
The number of pages that have been paged out.

faults
The number of times the **vm_fault** routine has been called.

cow_faults
The number of copy-on-write faults.

lookups
The number of object cache lookups.

hits
The number of object cache hits.

NOTES

This structure is machine word length specific because of the memory sizes returned.

RELATED INFORMATION

Functions: **task_info**, **host_statistics**, **host_page_size**.

Data Structures: **task_basic_info**.

APPENDIX H

Co–Location Data Structures

This appendix discusses the specifics of the various structures used as a part of the kernel's co-location server interfaces.

Status:

Experimental: [OSF-RI MK6]

mig_stub_routine

Typedef — Defines the calling convention for the MIG-generated server stubs

SYNOPSIS

```
[1]  typedef void (*mig_stub_routine) (
[2]            mach_msg_header_t          *InHeadP;
[3]            mach_msg_header_t          *OutHeadP);


[4]  typedef mig_stub_routine_t           mig_impl_routine_t;
[5]  typedef kern_return_t                (*mig_impl_routine_t)(void);
```

DESCRIPTION

The **mig_stub_routine** type is the definitionfor MIG-generated server stub routines. These routines: unpack the request message, call the server procedure, and pack the reply message.

FIELDS

InHeadP
: Pointer to the header for the request message.

OutHeadP
: Pointer to the header for the reply message.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_descriptor**.

routine_arg_descriptor

Status:

Experimental: [OSF-RI MK6]

Structure — Defines the nature of each argument in the routine call

SYNOPSIS

```
typedef enum {
        ROUTINE_ARG_PORT,
        ROUTINE_ARG_FIXED_ARRAY,
        ROUTINE_ARG_VAR_ARRY
} routine_arg_type;
typedef enum {
        ROUTINE_ARG_IN,
        ROUTINE_ARG_OUT,
        ROUTINE_ARG_INOUT
} routine_arg_direction;
typedef unsigned int                    routine_arg_offset;


struct routine_arg_descriptor
{
        routine_arg_type                type;
        routine_arg_direction           direction;
        routine_arg_offset              offset;
} *routine_arg_descriptor_t;
```

DESCRIPTION

The **routine_arg_descriptor** structure is the definition for the handling of each argument passed to/from a routine.

FIELDS

type
: The type of argument (Port, Array, etc.).

direction
: Whether this is an inout, ouput, or inout argument.

offset
: Offset in the list of routine arguments.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_descriptor, rpc_subsystem**.

Status:

Experimental: [OSF-RI MK6]

routine_descriptor

Structure — Defines the nature of each subsystem routine

SYNOPSIS

```
[1]  structure routine_descriptor
[2]  {
[3]          mig_impl_routine_t              impl_routine;
[4]          mig_stub_routine_t              stub_routine;
[5]          unsigned int                    argc;
[6]          unsigned int                    descr_count;
[7]          struct routine_arg_descriptor   *arg_descr;
[8]          unsigned int                    max_reply_msg;
[9]  } *routine_descriptor_t;
```

DESCRIPTION

The **routine_descriptor** structure is the definition of each exported function within a subsystem. The structure contains information of the function location, message disassembly routine, and the nature of the argument signature.

FIELDS

impl_routine
Server function pointer (code entry point).

stub_routine
Unmarshalling function pointer.

argc
Number of argument words.

descr_count
Number of complex argument descriptors.

arg_descr
Pointer to the beginning of the argument descriptor array (containing *descr_count* entries).

max_reply_msg
The maximum size for the reply message.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_arg_descriptor, rpc_subsystem**.

rpc_subsystem

Status:

Experimental: [OSF-RI MK6]

Structure — Defines the set of server routines in a subsystem

SYNOPSIS

```
struct rpc_subsystem
{
        struct subsystem                    *subsystem;
        mach_msg_id_t                       start;
        mach_msg_id_t                       end;
        unsigned int                        maxsize;
        vm_address_t                        base_addr;
        struct routine_descriptor           routine[1];
        struct routine_arg_descriptor       arg_descriptor[1];
} *rpc_subsystem_t;


#define RPC_SUBSYSTEM_NULL          ((rpc_subsystem_t) 0)
```

DESCRIPTION

The **rpc_subsystem** structure describes the set of server routines that can be invoked by mach_rpc() on the ports that are registered with the subsystem. For each routine, the routine number is given, along with the address of the implementation function in the server and a description of the arguments of the routine (i.e. it's signature).

This structure definition is only a template for what is really a variable-length structure (generated by MIG for each subsystem). The actual structures do not always have one entry in the *routine* array, and also have a varying number of entries in the arg_descr array. Each routine has an array of zero or more arg descriptors one for each complex arg. These arrays are all catenated together to form the *arg_descriptor* field of the subsystem struct. The arg_descr field of each routine entry points to a unique sub-sequence within this catenated array. The goal is to keep everything contiguous.

FIELDS

subsystem
: Reserved for system use.

start
: Minimum routine number.

end
: Maximum routine number (+1).

maxsize
: Maximum size of a message received from **mach_msg**.

base_addr
: The address of this **rpc_subsystem** structure in user space.

routine
: Starting point for the array of routin descriptors. The number of entires is equal to *start-end*.

arg_descriptor
: Starting point for the complex argument descriptors. The number of entires is equal to the sum of the descr_count fields for all of the routines.

RELATED INFORMATION

Functions: **mach_subsystem_create, mach_port_allocate_subsystem, mach_msg.**

Structures: **routine_arg_descriptor, routine_descriptor**.

APPENDIX I Error Return Values

This appendix lists the various kernel return values.

Error Code Format

An error code has the following format:

- system code (6 bits). The **err_get_system** (*err*) macro extracts this field.
- subsystem code (12 bits). The **err_get_sub** (*err*) macro extracts this field.
- error code (14 bits). The **err_get_code** (*err*) macro extracts this field.

The various system codes are:

- *err_kern* —kernel
- *err_us* — user space library
- *err_server*— user space servers
- *err_mach_ipc* — 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

MIG Stub Errors

MIG_ARRAY_TOO_LARGE

User specified array not large enough to hold returned array

MIG_BAD_ARGUMENTS

Message receiver found an invalid message size, invalid header fields or invalid descriptors. This could only happen if an invalidly formatted message (i.e., one that did not pass through Mach IPC) were passed to a MIG de-multiplexing routine.

MIG_BAD_ID

Bad message ID. This is only returned by MIG de-multiplexing routines when the message ID in the supplied message is not handled by that routine.

MIG_EXCEPTION

A server has raised an exception. [added from OSF/1 R1.3]

MIG_REMOTE_ERROR

A server has detected an error. [added from OSF/1 R1.3]

MIG_REPLY_MISMATCH

The message ID in a reply message is not 100 more than the message ID of the request message.

MIG_SERVER_DIED

Message recipient no longer exists, or the recipient destroyed the request message without replying.

MIG_TRAILER_ERROR

The message trailer has an unknown format. [added from OSF/1 R1.3]

MIG_TYPE_ERROR

The wrong number or size of data or rights was received.

Base IPC Status

MACH_MSG_SUCCESS

Normal IPC success. This is the same value as KERN_SUCCESS.

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_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.

IPC Send Errors

The following errors can occur when **mach_msg** is used with the MACH_SEND_MSG option. This is also the case for all function interfaces.

MACH_SEND_INTERRUPTED

Message send interrupted.

MACH_SEND_INVALID_DATA

Message buffer is unreadable.

MACH_SEND_INVALID_DEST

The destination port name in the message is MACH_PORT_NULL, MACH_PORT_DEAD, names a null or dead right, names a port set or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_SEND_INVALID_HEADER

A field in the message header had a bad value.

MACH_SEND_INVALID_MEMORY

An out-of-line memory region does not exist in the address space or is not readable.

MACH_SEND_INVALID_NOTIFY

The notify port name (MACH_SEND_CANCEL) specified to **mach_msg** is not a receive right.

MACH_SEND_INVALID_REPLY

The reply port name in the message is MACH_PORT_DEAD, names a null right, names a port set or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_SEND_INVALID_RIGHT

A port name in the message body is MACH_PORT_DEAD, names a null right, names a port set or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_SEND_INVALID_TRAILER

The trailer to be sent does not correspond to the current kernel format, or the sending task does not have the privilege to supply the message attributes.

MACH_SEND_INVALID_TYPE

An invalid msg-type specification was encountered. [added from OSF/1 R1.3]

MACH_SEND_MSG_TOO_SMALL

Message buffer doesn't contain a complete message.

MACH_SEND_NO_BUFFER

No kernel message buffer is available.

MACH_SEND_TIMED_OUT

Message not sent before time-out expired.

IPC Receive Errors

The following errors can be returned by **mach_msg** when used with the MACH_-RCV_MSG option. They can occur for kernel function interfaces.

MACH_RCV_BODY_ERROR

Error receiving kernel message body. See special bits.

MACH_RCV_HEADER_ERROR

Error receiving message header. See special bits.

MACH_RCV_IN_SET

The receive port name specified to **mach_msg** is a member of a port set.

MACH_RCV_INTERRUPTED

A software interrupt occurred.

MACH_RCV_INVALID_DATA

The message buffer was not writable.

MACH_RCV_INVALID_NAME

The receive port name specified to **mach_msg** is MACH_PORT_NULL, MACH_PORT_DEAD, names a null or dead right or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_RCV_INVALID_NOTIFY

The notify port name (MACH_RCV_NOTIFY) specified to **mach_msg** is not a receive right.

MACH_RCV_INVALID_TRAILER

The trailer type desired, or the number of trailer elements desired, is not supported by the kernel.

MACH_RCV_INVALID_TYPE

One or more scatter list descriptors did not have the type matching the corresponding incoming message descriptor or had an invalid copy (disposition) field.

MACH_RCV_LIMITS

The combined size of all out-of-line regions, the total of all port rights or the total message size exceeds the transfer limit for the port.

MACH_RCV_OVERWRITE_ERROR

A region described by a receive overwrite descriptor does not exist or cannot be written.

MACH_RCV_PORT_CHANGED

Receive right specified to **mach_msg** was moved into a set during the receive.

MACH_RCV_PORT_DIED

Receive right (or set) specified to **mach_msg** was sent away/died during receive.

MACH_RCV_SCATTER_SMALL

One or more scatter list descriptors specified an overwrite region smaller than the corresponding incoming region.

MACH_RCV_TIMED_OUT

A message was not received within the time-out value.

MACH_RCV_TOO_LARGE

Message buffer is not large enough for the message.

Generic Kernel Errors

KERN_SUCCESS

Successful completion

KERN_INVALID_ARGUMENT

The function requested was not applicable to this type of object.

KERN_INVALID_CAPABILITY

The supplied right is dead, null or not of the proper type.

KERN_INVALID_VALUE

A parameter's value was out of range (or possibly ill-formed). Specific error return values are returned if the parameter's value is properly formed and in range, but not a usable value at this time.

KERN_RESOURCE_SHORTAGE

A system resource could not be allocated to fulfill this request. This failure may not be permanent.

Port Manipulation Errors

KERN_INVALID_NAME

The port name doesn't denote a right in the task.

KERN_INVALID_RIGHT

The port name denotes a right, but not an appropriate right.

KERN_INVALID_TASK

The target task is not an active task. [added from OSF/1 R1.3]

KERN_NAME_EXISTS

The port name already denotes a right in the task.

KERN_NO_ACCESS

A bogus access restriction was encountered. [added from OSF/1 R1.3]

KERN_NO_SPACE

The task's port name space is full.

KERN_NOT_DEPRESSED

A **thread_depress_abort** was requested on a thread which was not currebtly depressed. [added from OSF/1 R1.3]

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. [added from OSF/1 R1.3]

KERN_RIGHT_EXISTS

The task already has send or receive rights for the port under another name.

KERN_UREFS_UNDERFLOW

The user-reference count would become negative.

KERN_UREFS_OVERFLOW

Operation would overflow limit on user-references.

Virtual Memory Manipulation Errors

KERN_INVALID_ADDRESS

Specified virtual address is not currently valid.

KERN_INVALID_OBJECT

The external memory manager failed to initialize the memory object. [added from OSF/1 R1.3]

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_MEMORY_PRESENT

An external memory manager attempted to supply pages already cached by the kernel.

KERN_NO_SPACE

The task's address space is full (not sufficient free space) or the specified address range is already in use.

KERN_PROTECTION_FAILURE

Specified memory is valid, but does not permit the required forms of access or the protection being requested exceeds that permitted.

Random Kernel Errors

KERN_ABORTED

The operation was aborted.

KERN_ALREADY_WAITING

A thread is attempting to wait for an event for which there is already a waiting thread.

KERN_DEFAULT_SET

An attempt was made to destroy the default processor set.

KERN_EXCEPTION_PROTECTED

An attempt was made to fetch an exception port that is protected, or to abort a thread while processing a protected exception.

KERN_FAILURE

A catch-all error for implementation dependent failures.

KERN_INVALID_HOST

An argument supplied to assert system privilege was not a host control port.

KERN_INVALID_LEDGER

A ledger was required but not supplied.

KERN_INVALID_MEMORY_CONTROL

The port was not a memory cache control port.

KERN_INVALID_POLICY

The specified scheduling policy is not currently enabled for the processor set.

KERN_INVALID_PROCESSOR_SET

An argument supplied to assert processor set privilege was not a processor set control port.

KERN_INVALID_SECURITY

An argument supplied to assert security privilege was not a host security port.

KERN_POLICY_LIMIT

The specified scheduling attributes exceeds the thread's limits.

Kernel Device Errors

D_SUCCESS

Normal device return. This is the same value as KERN_SUCCESS.

D_ALREADY_OPEN

Exclusive-use device already open.

D_DEVICE_DOWN

Device has been shut down.

D_INVALID_OPERATION

No filter port was specified.

D_INVALID_RECNUM

Invalid record (block) number.

D_INVALID_SIZE

Invalid IO size.

D_IO_ERROR

Hardware IO error.

D_NO_MEMORY

A memory allocation related to an I/O operation has failed. [added from OSF/1 R1.3]

D_NO_SUCH_DEVICE

No device with that name, or the device is not operational.

D_OUT_OF_BAND

Out-of-band condition occurred on device (such as typing control-C).

D_READ_ONLY

Data cannot be written to this device.

D_WOULD_BLOCK

Operation would block, but D_NOWAIT set.

APPENDIX J

Object Index

A

B

APPENDIX K

Interface and Structure Index