OSF RI 1994 Mach Kernel Interfaces *Draft*

Open Software Foundation and
Carnegie Mellon University

Marc J. Teller, Editor

August 11, 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

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 editted by Keith Loepere.

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.

do_mach_notify_dead_name

Status:

Believed to be stable

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**.

Status:

Believed to be stable

mach_port_insert_right

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 ob-

ject 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**.

memory_object_supply_completed

Status:

Believed to be stable

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**.

memory_object_synchronize

Status:

Believed to be stable

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**.

memory_object_synchronize_completed

Status:

Believed to be stable

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**.