DTOS Mach Kernel Interfaces

Secure Computing Corporation

Derived from the OSF Mach 3.0 Kernel Interfaces Document, Edited by Keith Loepere
This work is derived from the Mach 3 Kernel Interfaces book is in the Open Software Foundation Mach 3 series.

Books in the OSF Mach 3 series:

**Mach 3 Kernel Principles**

**Mach 3 Kernel Interfaces**

**Mach 3 Server Writer’s Guide**

**Mach 3 Server Library Interfaces**

Revision History:

Revision 2  MK67: January 7, 1992  OSF / Mach release
Revision 2.2  NORMA-MK12: July 15, 1992
Revision 2.3  NORMA-MK14: November 20, 1992

Revision 1.0  DTOS-MK01: December 28, 1993

Change bars indicate changes since NORMA-MK14:

Copyright 1992 by the Open Software Foundation, Inc. and Carnegie Mellon University.

Copyright 1997 by Secure Computing Corporation.

All rights reserved.

This document is derived from the OSF Revision 2.3 document. The following notations are provided un-
changed from that OSF baseline.

Permission to reproduce this document without fee is hereby granted, provided that the copies are not made
or distributed for direct commercial advantage, and the copyright notice and this permission notice appear in
all copies, derivative works or modified versions.

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

CHAPTER 1  Introduction .............................................. 1
  Interface Descriptions ........................................... 1
  Interface Types ................................................... 2
  Special Forms ...................................................... 3
  Parameter Types ................................................... 3
  Error Return Values .............................................. 4
  Security Controls ............................................... 5

CHAPTER 2  IPC Interface ............................................ 7
  mach_msg/mach_msg_secure ...................................... 8
  mach_msg_receive ............................................... 26
  mach_msg_send .................................................. 27

CHAPTER 3  Port Manipulation Interface .............................. 29
  do_mach_notify_dead_name ..................................... 30
  do_mach_notify_msg_accepted ................................. 32
  do_mach_notify_no_senders ................................... 34
  do_mach_notify_port_deleted ................................ 36
  do_mach_notify_port_destroyed ............................... 38
  do_mach_notify_send_once .................................... 40
  mach_port_allocate/mach_port_allocate_secure ............. 41
  mach_port_allocate_name/mach_port_allocate_name_secure .. 44
  mach_port_deallocate ......................................... 47
  mach_port_destroy ............................................. 48
  mach_port_extract_right ..................................... 50
  mach_port_get_receive_status ................................ 52
  mach_port_get_refs ............................................ 54
  mach_port_get_set_status ..................................... 56
  mach_port_insert_right ...................................... 58
  mach_port_mod_refs ........................................... 60
  mach_port_move_member ....................................... 62
  mach_port_names .............................................. 64
  mach_port_rename ............................................. 66
  mach_port_request_notification .............................. 68
  mach_port_set_mscount ............................... . . . . . . . . . . 71
  mach_port_set qlimit ......................................... 73
  mach_port_set_seqno .......................................... 75
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Interface</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Virtual Memory Interface</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>vm_allocate/vm_allocate_secure</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>vm_copy</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>vm_deallocate</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>vm_inherit</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>vm_machine_attribute</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>vm_map</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>vm_protect</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>vm_read</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>vm_region/vm_region_secure</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>vm_statistics</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>vm_wire</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>vm_write</td>
<td>107</td>
</tr>
<tr>
<td>5</td>
<td>External Memory Management Interface</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>memory_object_change_attributes</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>memory_object_change_completed</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>memory_object_copy</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>memory_object_data_error</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>memory_object_data_provided</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>memory_object_data_request</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>memory_object_data_return</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>memory_object_data_supply</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>memory_object_data_unavailable</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>memory_object_data_unlock</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>memory_object_data_write</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>memory_object_destroy</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>memory_object_get_attributes</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>memory_object_init</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>memory_object_lock_completed</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>memory_object_lock_request</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>memory_object_ready</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>memory_object_set_attributes</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>memory_object_supply_completed</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>memory_object_terminate</td>
<td>153</td>
</tr>
<tr>
<td>6</td>
<td>Thread Interface</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>catch_exception_raise</td>
<td>156</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>Processor Management and Scheduling Interface</td>
<td>231</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>host_processor_set_priv.</td>
<td>232</td>
</tr>
<tr>
<td></td>
<td>host_processor_sets</td>
<td>233</td>
</tr>
<tr>
<td></td>
<td>host_processors</td>
<td>235</td>
</tr>
<tr>
<td></td>
<td>processor_assign</td>
<td>236</td>
</tr>
<tr>
<td></td>
<td>processor_control</td>
<td>238</td>
</tr>
<tr>
<td></td>
<td>processor_exit</td>
<td>240</td>
</tr>
<tr>
<td></td>
<td>processor_get_assignment</td>
<td>241</td>
</tr>
<tr>
<td></td>
<td>processor_info</td>
<td>242</td>
</tr>
<tr>
<td></td>
<td>processor_set_create</td>
<td>244</td>
</tr>
<tr>
<td></td>
<td>processor_set_default</td>
<td>246</td>
</tr>
<tr>
<td></td>
<td>processor_set_destroy</td>
<td>247</td>
</tr>
<tr>
<td></td>
<td>processor_set_info</td>
<td>248</td>
</tr>
<tr>
<td></td>
<td>processor_set_max_priority</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>processor_set_policy_disable</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>processor_set_policy_enable</td>
<td>254</td>
</tr>
<tr>
<td></td>
<td>processor_set_tasks</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>processor_set_threads</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>processor_start</td>
<td>257</td>
</tr>
<tr>
<td></td>
<td>task_assign</td>
<td>258</td>
</tr>
<tr>
<td></td>
<td>task_assign_default</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>task_get_assignment</td>
<td>262</td>
</tr>
<tr>
<td></td>
<td>task_priority</td>
<td>263</td>
</tr>
<tr>
<td></td>
<td>thread_assign</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td>thread_assign_default</td>
<td>266</td>
</tr>
<tr>
<td></td>
<td>thread_get_assignment</td>
<td>267</td>
</tr>
<tr>
<td></td>
<td>thread_max_priority</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>thread_policy</td>
<td>270</td>
</tr>
<tr>
<td></td>
<td>thread_priority</td>
<td>271</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 10</th>
<th>Kernel Device Interface</th>
<th>273</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>device_close</td>
<td>274</td>
</tr>
<tr>
<td></td>
<td>device_get_status</td>
<td>275</td>
</tr>
</tbody>
</table>
### device_map

- device_map ........................................ 277
- device_open ...................................... 279
- device_read ....................................... 282
- device_read_inband .............................. 285
- device_set_filter ................................. 288
- device_set_status ................................. 292
- device_write ..................................... 294
- device_write_inband ............................. 297
- evc_wait .......................................... 300

### CHAPTER 11 Security Server Interface

- avc_cache_control, avc_cache_control_trap ........ 304
- extract_aid ....................................... 306
- extract_mid ....................................... 307
- make_sid .......................................... 308
- SSI_compute_access_vector ....................... 309
- SSI_context_to_mid ................................ 312
- SSI_load_security_policy ....................... 314
- SSI_record_name_server .......................... 315
- SSI_register_caching_server .................... 316
- SSI_short_context_to_mid ....................... 318
- SSI_mid_to_context ................................ 320
- SSI_mid_to_short_context ....................... 322
- SSI_transfer_security_server_ports ............ 324
- SSI_transition_domain ............................ 326

### APPENDIX A MIG Server Routines

- device_reply_server ............................. 330
- exc_server ....................................... 332
- memory_object_default_server .................. 334
- memory_object_server ............................ 336
- notify_server ...................................... 338
- prof_server ....................................... 340
- seqnos_memory_object_default_server .......... 341
- seqnos_memory_object_server .................. 343
- seqnos_notify_server ............................. 345

### APPENDIX B Default Memory Management Interface

- default_pager_info ............................... 348
- default_pager_object_create .................... 350
processor_basic_info .................................. 418
processor_set_basic_info ............................. 419
processor_set_sched_info ......................... 420
sampled_pc ........................................ 421
security_id_t ..................................... 422
task_basic_info ................................... 423
task_basic_secure_info ............................ 424
task_thread_times_info ............................ 426
thread_basic_info ................................ 427
thread_sched_info ................................ 429
time_value ......................................... 431
vm_statistics ...................................... 432

APPENDIX F  Error Return Values ...................... 435
Error Code Format .................................. 435
MIG Stub Errors .................................... 436
Base IPC Status .................................... 436
IPC Send Errors ..................................... 437
IPC Receive Errors ................................ 438
Generic Kernel Errors .............................. 439
Port Manipulation Errors ......................... 440
Virtual Memory Manipulation Errors ............ 441
Random Kernel Errors ............................. 441
Kernel Device Errors .............................. 442

APPENDIX G  Permission Definitions .................. 445
Device Port Permissions ............................ 445
Host Privilege Port Permissions ................. 446
Host Port Permissions ............................. 447
Kernel Reply Port Permissions .................... 449
Memory Object Permissions ....................... 449
Memory Control Port Permissions ............... 449
Processor Port Permissions ....................... 451
Processor Set Permissions ......................... 451
Task Port Permissions ............................ 452
Thread Port Permissions .......................... 457
IPC Permissions ................................... 459
CHAPTER 1  

Introduction

This book documents the various interfaces to the DTOS variant of the Mach 3 kernel. The text generally describes each interface to the kernel in isolation. Entries that have a special security relevant variant are described together to avoid redundancy. 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
Introduction

- 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
- Identification of the request specific security permissions that must be held to make the request
- 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 three 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 may fail because required security permissions are not held by the requesting task. The list of error returns for these calls should always be considered to also include the security related error, KERN_INSUFFICIENT_PERMISSION.
- These calls only invoke their expected effect when the acting port is indeed a port of the specified type. That is, if a call expects a port that names a task (a kernel task port) and the port is instead a port managed by a task, the 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 Interfaces**. For any given kernel message, there are any number of possible server interface calling sequences that can be generated, by permuting the order of the data provided in the message, omitting some data elements or including or omitting various header field elements (such as sequence numbers). In most cases, a single server interface calling sequence has been chosen with a given MIG generated server message de-multiplexing routine that calls these interfaces. In some cases, there are more than one MIG generated server routines which call upon different server interfaces associated with that MIG service routine. In any event, all **Server Interfaces** contain within their documentation the name of the MIG generated server routine that invokes the interface.

### Special Forms

There are various special interface forms defined in this volume.

- The **Macro** form specifies macros (typically defined in `mach.h`) that provide shorthand equivalents for some variations of the longer function call.

- The **Sequence Number** form of a **Server Interface** defines an additional MIG generated interface that supplies the sequence number from the message causing the server interface to be invoked. The existence of such a form implies the existence of an alternate MIG generated message de-multiplexing routine that invokes this special interface form.

- The **Asynchronous** form defines a MIG generated version of a **Function** that allows the function to be invoked asynchronously. Such a version requires an additional parameter to specify the reply port to which the reply is sent. The return value from the asynchronous function is the return status from the `mach_msg` call sending the request, not the resulting status of the kernel operation. The asynchronous interface also requires a matching **Server Interface** that defines the reply message containing data that would have been output values from the normal function, as well as the resulting status from the kernel operation.

### Parameter Types

Each interface description supplies the C type of the various parameters. The parameter descriptions then indicate whether these parameters are input (“in”), output (“out”) or both (“in/out”). This information appears in square brackets before the parameter description. Additional information also appears within these brackets for special or non-obvious parameter conventions.
The most common notation is “scalar”, which means that the parameter somehow derives from an \textit{int} type. Port types are scalar types but are marked specifically as to the type of port named by the parameter.

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

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 \texttt{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 \texttt{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 \texttt{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 \texttt{vm\_deallocate}d when the data is no longer needed.

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

\section*{Error Return Values}

APPENDIX F 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 messaging and security 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.

**Security Controls**

All of the MIG generated and the hybrid MIG/system call interfaces are subject the following general control rules.

- The requesting task must have av_send permission to the first port in the parameter list.
- The requesting task must also have av_transfer_send and av_set_reply permission to the reply port provided in the MIG generated request message.
- All IPC permission checks are applied to MIG generated interfaces. ie if a port is given as an output parameter, the client must have av_hold_send, and av_can_send permissions.

Thus the respective security sections of each interface description, only describes the control issues specific to that interface.
Introduction

In the case of the “pure” system call interfaces only the interface specific control check is made. In this case the check is made against the implicit task or thread port as is appropriate for the interface.
This chapter discusses the specifics of the kernel’s inter-"process" communication (IPC) interfaces. The interfaces discussed are only the interfaces directly involved in sending and receiving IPC messages.
**mach_msg/mach_msg_secure**

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

**SYNOPSIS**

```c
mach_msg_return_t mach_msg (mach_msg_header_t* msg,
                             mach_msg_option_t option,
                             mach_msg_size_t send_size,
                             mach_msg_size_t rcv_size,
                             mach_port_t rcv_name,
                             mach_msg_timeout_t timeout,
                             mach_port_t notify);

mach_msg_return_t mach_msg_secure (mach_msg_header_t* msg,
                                    mach_msg_option_t option,
                                    mach_msg_size_t send_size,
                                    mach_msg_size_t rcv_size,
                                    mach_port_t rcv_name,
                                    mach_msg_timeout_t timeout,
                                    mach_port_t notify,
                                    security_id_t* rec_subj_sid,
                                    security_id_t* sender_subj_sid,
                                    int av_buf_size,
                                    mach_access_vector_t* av_buf);
```

**DESCRIPTION**

The **mach_msg** and **mach_msg_secure** system calls send and receive Mach messages. Mach messages contain typed data, which can include port rights and addresses of large regions of memory.

If the `option` argument contains MACH_SEND_MSG, it sends a message. The `send_size` argument specifies the size of the message to send. The `msggh_remote_port` field of the message header specifies the destination of the message.

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

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

If the option argument specifies neither MACH_SEND_MSG nor MACH_RCV_MSG, then mach_msg and mach_msg_secure do 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.

SECURITY

The DTOS kernel provides controls beyond those of the Mach capability mechanism described in the NOTES section below. The kernel security mechanisms enforce the permissions described in the mach_access_vector_t structure defined in APPENDIX E. In addition to the appropriate rights, the following access permissions control message operations.

Sending Message
The sending task must have av_can_send permission to the destination port. If a reply port is used, the sending task must have av_set_reply permission to the reply port.

Receiving Message
The receiving task must have av_can_receive permission to the port indicated by rcv_name. Messages will be received from a port in a port set only if the requesting task has av_can_receive permission to the port. When a task uses a port as a reply port for an RPC type of operation, the requesting task must also have av_can_send permission to that port.

Passing SEND, SEND_ONCE or RECEIVE Right
Passing of rights is done by sending a message to a port P1 where the body of the message contains a port right to port P2. The task sending the message must have respectively, av_transfer_send, av_transfer_send_once or av_transfer_receive permission to port P2, depending on whether the right is a send, send_once or receive. In addition the sending task must have av_transfer_right to the destination port P1, in order to transfer any right in the body of the message.

Upon receipt of a right the receiving task must have respectively, av_hold_send, av_hold_send_once or av_hold_receive permission to the port associated with the right in the message body.

Passing Out Of Line data
To pass out of line data in a message the sending task must have av_transfer_ool permission to the destination port. In addition, if the out-of-line data contains a port right, the permission requirements
IPC Interface

DESCRIBED IN THE ABOVE SECTION ON “PASSING SEND, SEND_ONCE OR RECEIVE RIGHT” ALSO APPLY.

THE SECURITY ASPECTS OF MACH_MSG AND MACH_MSG_SECURE INCLUDE THE FOLLOWING ADDITIONAL CONTROL ISSUES.

- ON A RECEIVE, THE RECEIVING TASK MUST HAVE AV_INTERPOSE PERMISSION TO RECEIVE MESSAGES DESIGNATED TO SUBJECT SECURITY IDENTIFIERS OTHER THAN THAT OF THE RECEIVING TASK.

THE SECURITY ASPECTS OF MACH_MSG_SECURE INCLUDE THE FOLLOWING ADDITIONAL CONTROL ISSUES.

- ON A SEND THE SENDING TASK MUST HAVE AV_SPECIFY PERMISSION TO THE DESTINATION PORT IN ORDER TO SPECIFY THE MESSAGE SENDER’S SUBJECT SECURITY IDENTIFIER TO BE ASSOCIATED WITH THE MESSAGE.
- ON A SEND THE SENDING TASK MUST HAVE AV_SPECIFY PERMISSION TO THE DESTINATION PORT IN ORDER TO SPECIFY ANY OF THE VALUES IN THE AV_BUF.

IF THE SENDING TASK DOES NOT SPECIFY OR DOES NOT HAVE AV_SPECIFY PERMISSION TO THE DESTINATION PORT THE DTOS KERNEL PROVIDES THE SECURITY IDENTIFIER OF THE SENDING TASK. IN ALL CASES THE DTOS KERNEL ASSOCIATES THE ACCESS VECTOR DESCRIBING THE SENDING TASKS PERMISSION TO THE DESTINATION PORT WITH THE MESSAGE.

IN-LINE AND OUT-OF-LINE DATA ARE CURRENTLY HANDLED DIFFERENTLY WITH RESPECT TO THE SECURITY IDENTIFIER ASSIGNED TO THE DATA. IN-LINE DATA IS ASSIGNED A SECURITY IDENTIFIER CORRESPONDING TO THE SECURITY IDENTIFIER OF THE MEMORY REGION WHERE IT IS PLACED. OUT-OF-LINE DATA MAY RETAINS THE SECURITY IDENTIFIER ASSIGNED TO THE MEMORY REGION FROM WHICH THE DATA CAME IF SO REQUESTED.

PARAMETERS

msg
[pointer to in/out structure containing random and reply ports] A message buffer. This should be aligned on a long-word boundary.

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

send_size
[in scalar] When sending a message, specifies the size of the message buffer. Otherwise zero should be supplied.

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

[in random port] 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 port] When using the MACH_SEND_NOTIFY, MACH_SEND_CANCEL, and MACH_RCV_NOTIFY options, specifies the port used for the notification. Otherwise MACH_PORT_NULL should be supplied.

**rec_subj_sid**

[pointer to in/out security id] When sending a message this parameter specifies the subject security identifier of the tasks that will be allowed to receive the message. Set to the address of a location that contains SEC_NULL_SID to indicate that there is no receiver restriction on the message.

When receiving a message this parameter contains the subject security identifier which the sender specified as the intended message receiver. Returns the address of a location containing SEC_NULL_SID if no intended recipient was supplied.

**sender_subj_sid**

[pointer to in/out security id] When sending a message this parameter specifies the subject security identifier to be provided as the message’s effective sender. The sender must have av_specify access to the port for the value to be used. Set to the address of a location containing SEC_NULL_SID to indicate that the sending task’s subject security identifier is to be used. When receiving a message this parameter contains the effective subject security identifier of the message sender.

**av_buf_size**

The size of the subsequent structure av_buf in bytes. If this size is set to zero, it is assumed that av_buf is not specified.

**av_buf**

[pointer to in/out access vector array structure] When receiving a message, this parameter points to a buffer that will contain the access vector describing the effective sender’s permission to the port providing the message, the notify vector, the override vector, and the cache control vector.
When sending a message, this parameter points to a buffer to the access vector, the notify vector, the override vector, and the cache control vector that the receiver will receive. The sender must have `av_specify` access to the port for the value to be used. Set to `MACH_NO_LABEL` to indicate that the effective sender’s permission is to be provided to the receiver.

**NOTES**

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

**Major Concepts**

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

The message primitives let tasks send and receive messages. Tasks send messages to ports. Messages sent to a port are delivered reliably (messages may not be lost) and are received in the order in which they were sent. Messages contain a fixed-size header and a variable amount of typed data following the header. The header describes the destination and size of the message.

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

Ports hold a queue of messages. Tasks operate on a port to send and receive messages by exercising capabilities (rights) for the port. Multiple tasks can hold send rights for a port. Tasks can also hold send-once rights, which grant the ability to send a single message. Only one task can hold the receive capability (receive right) for a port. Port rights can be transferred between tasks via messages. The sender of a message can specify in the message body that the message contains a port right. If a message contains a receive right for a port, then the receive right is removed from the sender of the message and the right is transferred to the receiver of the message. While the receive right is in transit, tasks holding send rights can still send messages to the port, and they are queued until a task acquires the receive right and uses it to receive the messages.
Tasks can receive messages from ports and port sets. The port set abstraction allows a single thread to wait for a message from any of several ports. Tasks manipulate port sets with a port set name, which is taken from the same name space as are the port rights. The port-set name may not be transferred in a message. A port set holds receive rights, and a receive operation on a port set blocks waiting for a message sent to any of the constituent ports. A port may not belong to more than one port set, and if a port is a member of a port set, the holder of the receive right can’t receive directly from the port.

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

Port Rights

Each task has its own space of port rights. Port rights are named with positive integers. Except for the reserved values MACH_PORT_NULL (0) and MACH_PORT_DEAD (-1), this is a full 32-bit name space. When the kernel chooses a name for a new right, it is free to pick any unused name (one which denotes no right) in the space.

There are three basic kinds of rights: receive rights, send rights and send-once rights. A port name can name any of these types of rights, a port-set, be a dead name, or name nothing. Dead names are not capabilities. They act as place-holders to prevent a name from being otherwise used.

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

Tasks may hold multiple user-references for send rights and dead names. When a task receives a send right which it already holds, the kernel increments the right’s user-reference count. When a task de-allocates a send right, the kernel decrements its user-reference count, and the task only loses the send right when the count goes to zero.

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

Each send-once right generated guarantees the receipt of a single message, either a message sent to that send-once right or, if the send-once right is in any way destroyed, a send-once notification.
A message body can carry port rights; the `msgt_name (msgtl_name)` field in a type descriptor specifies the type of port right and how the port right is to be extracted from the caller. The values MACH_PORT_NULL and MACH_PORT_DEAD are always valid in place of a port right in a message body.

In a sent message, the following `msgt_name` values denote port rights:

**MACH_MSG_TYPE_MAKE_SEND**
The message will carry a send right, but the caller must supply a receive right. The send right is created from the receive right, and the receive right’s make-send count is incremented.

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

**MACH_MSG_TYPE_MOVE_SEND**
The message will carry a send right, and the caller should supply a send right. The user reference count for the supplied send right is decremented, and the right is destroyed if the count becomes zero. Unless a receive right remains, the name becomes available for recycling. The caller may also supply a dead name, which loses a user reference, and the receiving task will get MACH_PORT_DEAD.

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

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

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

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

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

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

MACH_MSG_TYPE_PORT_RECEIVE
This value is an alias for MACH_MSG_TYPE_MOVE_RECEIVE. The message carried a receive right. If the receiving task already has send rights for the port, then that name for the port will be reused. Otherwise, the right will have a new, previously unused, name. The make-send count and sequence number of the receive right are reset to zero, but the port retains other attributes like queued messages, extant send and send-once rights, and requests for port-destroyed and no-senders notifications. (Note: It is currently planned to remove port-destroyed notifications from the kernel interface and to define no-senders notifications as being canceled when a receive right is moved.)

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

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

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

Out-of-line memory arrives somewhere in the receiver’s address space as new memory. It has the same inheritance and protection attributes as newly vm_allocate ed memory. The receiver has the responsibility of de-allocating (with vm_deallocate) the memory when it is no longer needed. Security-conscious receivers should exercise caution when dealing with out-of-line memory.
from un-trustworthy sources, because the memory may be backed by an unreliable memory manager.

Null out-of-line memory is legal. If the out-of-line region size is zero (for example, because msgtl_number is zero), then the region’s specified address is ignored. A received null out-of-line memory region always has a zero address.

Unaligned addresses and region sizes that are not page multiples are legal. A received message can also contain regions with unaligned addresses and funny sizes. In the general case, the first and last pages in the new memory region in the receiver do not contain data from the sender, but are partly zero. The received address points into the middle of the first page. This possibility doesn’t complicate de-allocation, because vm_deallocate does the right thing, rounding the start address down and the end address up to de-allocate all arrived pages.

Out-of-line memory has a de-allocate option, controlled by the msgt_deallocate bit. If it is TRUE and the out-of-line memory region is not null, then the region is implicitly de-allocated from the sender, as if by vm_deallocate. In particular, the start and end addresses are rounded so that every page overlapped by the memory region is de-allocated. The use of msgt_deallocate effectively changes the memory copy into a memory movement. In a received message, msgt_deallocate is TRUE in type descriptors for out-of-line memory.

Out-of-line memory can carry port rights.

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

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

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

These options modify MACH_SEND_MSG. If MACH_SEND_MSG is not also specified, they are ignored.
MACH_SEND_TIMEOUT
The timeout argument should specify a maximum time (in milliseconds) for the call to block before giving up. If the message can’t be queued before the timeout interval elapses, then the call returns MACH_SEND_TIMED_OUT. A zero timeout is legitimate.

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

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

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

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

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

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

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

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

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

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

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

**Message Receive**

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

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

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

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

The destination and reply ports are reversed in a received message header. The `msg_only_local_port` field carries the name of the destination port, from which the message was received, and the `msg_only_remote_port` field carries the reply port
right. The bits in *msgh_bits* are also reversed. The MACH_MSGH_BITS_LOCAL bits have the value MACH_MSG_TYPE_PORT_SEND if the message was sent to a send right, and the value MACH_MSG_TYPE_PORT_SEND_ONCE if was sent to a send-once right. The MACH_MSGH_BITS_REMOTE bits describe the reply port right.

Received messages are stamped with a sequence number, taken from the port from which the message was received. (Messages received from a port set are stamped with a sequence number from the appropriate member port.) Newly created ports start with a zero sequence number, and the sequence number is reset to zero whenever the port's receive right moves between tasks. When a message is de-queued from the port, it is stamped with the port's sequence number and the port's sequence number is then incremented. The de-queue and increment operations are atomic, so that multiple threads receiving messages from a port can use the *msgh_seqno* field to reconstruct the original order of the messages.

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

In almost all cases, *msgh_local_port* will specify the name of a receive right, either *rcv_name*, or, if *rcv_name* is a port set, a member of *rcv_name*. If other threads are concurrently manipulating the receive right, the situation is more complicated. If the receive right is renamed during the call, then *msgh_local_port* specifies the right's new name. If the caller loses the receive right after the message was de-queued from it, then *mach_msg* will proceed instead of returning MACH_RCV_PORT_DIED. If the receive right was destroyed, then *msgh_local_port* specifies MACH_PORT_DEAD. If the receive right still exists, but isn't held by the caller, then *msgh_local_port* specifies MACH_PORT_NULL.

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

**MACH_RCV_TIMEOUT**

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

**MACH_RCV_NOTIFY**

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

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

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

If a resource shortage prevents the reception of a port right, the port right is destroyed and the caller sees the name MACH_PORT_NULL. If a resource shortage prevents the reception of an out-of-line memory region, the region is destroyed and the caller sees a zero address. In addition, the `msgt_size` (msgtl_size) field in the region's type descriptor is changed to zero. If a resource shortage prevents the reception of out-of-line memory carrying port rights, then the port rights are always destroyed if the memory region can not be received. A task never receives port rights or memory for which it is not told.

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

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

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

For example, consider sending a message with the destination port specified as MACH_MSG_TYPE_MOVE_SEND and the reply port specified as MACH_MSG_TYPE_COPY_SEND. The same send right, with one user-reference, is supplied for both the `msgh_remote_port` and `msgh_local_port` fields. Because `mach_msg` processes the message header atomically, this succeeds. If
msg\_remote\_port were processed before \texttt{msg\_local\_port}, then \texttt{mach\_msg} would return \texttt{MACH\_SEND\_INVALID\_REPLY} in this situation.

On the other hand, suppose the destination and reply port are both specified as \texttt{MACH\_MSG\_TYPE\_MOVE\_SEND}, and again the same send right with one user-reference is supplied for both. Now the send operation fails, but because it processes the header atomically, \texttt{mach\_msg} can return either \texttt{MACH\_SEND\_INVALID\_DEST} or \texttt{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 \texttt{MACH\_RCV\_PORT\_DIED}. If the de-allocation happens after the receive, then the \texttt{msg\_local\_port} and the \texttt{msg\_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 \texttt{msg\_local\_port} and \texttt{msg\_remote\_port} fields both specify \texttt{MACH\_PORT\_DEAD}. Because the header is processed atomically, it is not possible for just one of the two fields to hold \texttt{MACH\_PORT\_DEAD}.

The \texttt{MACH\_RCV\_NOTIFY} option provides a more likely example. Suppose a message carrying a send-once right reply port is received with \texttt{MACH\_RCV\_NOTIFY} at the same time the reply port is destroyed. If the reply port is destroyed first, then \texttt{msg\_remote\_port} specifies \texttt{MACH\_PORT\_DEAD} and the kernel does not generate a dead-name notification. If the reply port is destroyed after it is received, then \texttt{msg\_remote\_port} specifies a dead name for which the kernel generates a dead-name notification. It is not possible to receive the reply port right and have it turn into a dead name before the dead-name notification is requested; as part of the message header the reply port is received atomically.

**Implementation**

\texttt{mach\_msg} and \texttt{mach\_msg\_secure} are wrappers for system calls. These routines have the responsibility for repeating the interrupted system call.

**CAUTIONS**

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

If \texttt{MACH\_RCV\_TIMEOUT} is used without \texttt{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. \texttt{MACH\_SEND\_TIMEOUT} without \texttt{MACH\_SEND\_INTERRUPT} suffers from the same problem.
RETURN VALUE

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

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

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

MACH_SEND_INVALID_DATA
The supplied message buffer was not readable.

MACH_SEND_INVALID_HEADER
The msgh_bits value was invalid.

MACH_SEND_INVALID_DEST
The msgh_remote_port value was invalid.

MACH_SEND_INVALID_REPLY
The msgh_local_port value was invalid.

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

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

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

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

MACH_SEND_INVALID_TYPE
A type descriptor was invalid.

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

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

MACH_SEND_TIMED_OUT
The timeout interval expired.
MACH_SEND_INTERRUPTED
A software interrupt occurred.

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

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

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

These return codes imply that the message was queued:

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

MACH_MSG_SUCCESS
The message was queued.

The receive operation can generate the following return codes. These return codes imply that the call did not de-queue a message:

MACH_RCV_INVALID_NAME
The specified rcv_name was invalid.

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

MACH_RCV_TIMED_OUT
The timeout interval expired.

MACH_RCV_INTERRUPTED
A software interrupt occurred.

MACH_RCV_PORT_DIED
The caller lost the rights specified by rcv_name.

MACH_RCV_PORT_CHANGED
rcv_name specified a receive right which was moved into a port set during the call.
MACH_RCV_TOO_LARGE
When using MACH_RCV_LARGE, and the message was larger than rcv_size. The message is left queued, and its actual size is returned in the msgh_size field of the message buffer.

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

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

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

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

These return codes imply that a message was received:

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

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

MACH_MSG_SUCCESS
A message was received.

Resource shortages can occur after a message is de-queued, while transferring port rights and out-of-line memory regions to the receiving task. In this situation, mach_msg and mach_msg_secure return MACH_RCV_HEADER_ERROR or MACH_RCV_BODY_ERROR. 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.

MACH_MSG_INSUFFICIENT_PERMISSION
A permission check failure prevented the reception of a port right.

RELATED INFORMATION
Functions: mach_msg_receive, mach_msg_send.

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

*Function* — Receives a message from a port or port set

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

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

**DESCRIPTION**

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

```
mach_msg(header, MACH_RCV_MSG, 0, header->msgh_size,
         header->msgh_local_port, MACH_MSG_TIMEOUT_NONE,
         MACH_PORT_NULL);
```

**SECURITY**

The receiving task must have `av_can_receive` permission to the port indicated by `rcv_name`. Messages will be received from a port in a port set only if the requesting task has `av_can_receive` permission to the port. When a task uses a port as a reply port for an RPC type of operation, the requesting task must also have `av_can_send` permission to that port.

**PARAMETERS**

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

**RETURN VALUE**

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

**RELATED INFORMATION**

Functions: `mach_msg`, `mach_msg_send`.

Data Structures: `mach_msg_header`, `mach_msg_type`, `mach_msg_type_long`. 
mach_msg_send

Function — Sends a message to a port

LIBRARY
Not declared anywhere.

SYNOPSIS

mach_msg_return_t mach_msg_send
            (mach_msg_header_t* header);

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

mach_msg(header, MACH_SEND_MSG, header->msgh_size, 0,
         MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE,
         MACH_PORT_NULL);

SECURITY
The sending task must have av_can_send permission to the destination port. If a
reply port is used, the sending task must have av_set_reply permission to the re-
ply port.

PARAMETERS

header
    [pointer to in structure containing random and reply ports] The address
    of the buffer that contains the message to be sent.

RETURN VALUE
Refer to mach_msg for a description of the send errors.

RELATED INFORMATION
Functions: mach_msg, mach_msg_receive.

Data Structures: mach_msg_header, mach_msg_type, mach_msg_type_long.
This chapter discusses the specifics of the kernel’s port manipulation interfaces. This includes port, port set and port right related functions. Also included are interfaces that return port related status information that applies to a single task.
do_mach_notify_dead_name

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

LIBRARY

Not declared anywhere.

SYNOPSIS

```c
kern_return_t do_mach_notify_dead_name
    (notify_port_t notify,
     mach_port_name_t name);
```

```c
kern_return_t do_seqnos_mach_notify_dead_name
    (notify_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.

SECURITY

There are no security limitations on this kernel outcall.

PARAMETERS

- **notify**
  [in notify port] 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.
do_mach_notify_dead_name

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

RETURN VALUE
Irrelevant.

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

Server Interface — Handles the occurrence of a message accepted notification

LIBRARY

Not declared anywhere.

SYNOPSIS

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

do_seqnos_mach_notify_msg_accepted
    Sequence Number form

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

DESCRIPTION

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

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

SECURITY

There are no security limitations on this kernel outcall.

PARAMETERS

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

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

- **name**
  - [in scalar] The port whose message was accepted.
do_mach_notify_msg_accepted

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

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

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

LIBRARY
Not declared anywhere.

SYNOPSIS

kern_return_t do_mach_notify_no_senders
    (notify_port_t notify,
     mach_port_mscount_t mscount);

do_seqnos_mach_notify_no_senders
Sequence Number form

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

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

SECURITY
There are no security limitations on this kernel outcall.

PARAMETERS

notify
    [in notify port] 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.

RETURN VALUE
Irrelevant.
do_mach_notify_no_senders

RELATED INFORMATION

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

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

LIBRARY
Not declared anywhere.

SYNOPSIS

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

```
do_seqnos_mach_notify_port_deleted
    Sequence Number form
    kern_return_t do_seqnos_mach_notify_port_deleted
        (notify_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.

SECURITY

There are no security limitations on this kernel outcall.

PARAMETERS

```
notify
    [in notify port] 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.
```
do_mach_notify_port_deleted

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

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

**Server Interface** — Handles the occurrence of a port destroyed notification

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

```c
kern_return_t do_mach_notify_port_destroyed
    (notify_port_t notify,
     mach_portReceive_t rights);
```

```c
do_seqnos_mach_notify_port_destroyed
Sequence Number form

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

**DESCRIPTION**

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

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

**SECURITY**

There are no security limitations on this kernel outcall.

**PARAMETERS**

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

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

- **rights**
  
  [in random port] The receive right that would have been destroyed.

**RETURN VALUE**

Irrelevant.
do_mach_notify_port_destroyed

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

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

LIBRARY

Not declared anywhere.

SYNOPSIS

    kern_return_t do_mach_notify_send_once
        (notify_port_t notify);

    do_seqnos_mach_notify_send_once
        Sequence Number form
        kern_return_t do_seqnos_mach_notify_send_once
            (notify_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.

SECURITY

There are no security limitations on this kernel outcall.

PARAMETERS

    notify
        [in notify port] 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, mach_msg, mach_port_request_notification,
do_mach_notify_msg_accepted, do_mach_notify_no_senders,
do_mach_notify_port_deleted, do_mach_notify_port_destroyed,
do_mach_notify_dead_name.
mach_port_allocate/mach_port_allocate_secure

**Function** — Creates a port right and optionally associates an object security identifier with the port.

**SYNOPSIS**

```c
kern_return_t mach_port_allocate
    (mach_port_t task,
     mach_port_right_t right,
     mach_port_t* name);

kern_return_t mach_port_allocate_secure
    (mach_port_t task,
     mach_port_right_t right,
     mach_port_t* name,
     security_id_t obj_sid);
```

**DESCRIPTION**

The mach_port_allocate function creates a new right in the specified task. The new right’s name is returned in name. The mach_port_allocate_secure function creates a new right in the specified task with the specified object security identifier.

**SECURITY**

The requesting task must hold tsv_add_name permission to the task port task. If the request results in a new receive right being created for task’s task, task must have av_hold_receive permission to the newly allocated port.

When using mach_port_allocate, the port is allocated with an object security identifier derived from task’s subject security identifier. Refer to the Software Design Document for further information on how SIDs are derived.

If mach_port_allocate_secure is given a SEC_NULL_SID as the obj_sid, then its behavior is essentially the same as mach_port_allocate.

**PARAMETERS**

- **task**
  
  [in task port] 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. An object security identifier cannot be associated with a port set, hence, if one is specified with `mach_port_allocate_secure`, it will be ignored.

MACH_PORT_RIGHT_DEAD_NAME

`mach_port_allocate` creates a dead name. The new dead name has one user reference. An object security identifier cannot be associated with a dead name, hence, if one is specified with `mach_port_allocate_secure`, it will be ignored.

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

`obj_sid` [in security id] The object security identifier to be associated with the created port. The interface `SSI_context_to_mid` can be used to obtain a mandatory identifier from the Security Server. The mandatory identifier and the authentication identifier can be combined into a security identifier via `make_sid`.

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_name, mach_port_allocate_name_secure, mach_port_deallocate, mach_port_insert_right, mach_port_extract_right, SSI_context_to_mid, make_sid.
**mach_port_allocate_name/**

**mach_port_allocate_name_secure**

Function — Creates a port right with a given name and optionally associates an object security identifier with the port.

**SYNOPSIS**

```c
kern_return_t mach_portAllocateName(mach_port_t task, mach_port_right_t right, mach_port_t name);
kern_return_t mach_portAllocateNameSecure(mach_port_t task, mach_port_right_t right, mach_port_t name, security_id_t obj_sid);
```

**DESCRIPTION**

The `mach_port_allocate_name` function creates a new right in the specified task, with a specified name for the new right. The `mach_port_allocate_name_secure` function creates a new right in the specified task, with a specified name and a specified object security identifier.

**SECURITY**

The requesting task must hold `tsv_add_name` permission to the task port `task`. If the request results in a new receive right being created for `task`’s task, task must have `av_hold_receive` permission to the newly allocated port.

When using `mach_port_allocate_name`, the port is allocated with an object security identifier derived from `task`’s subject security identifier.

If `mach_port_allocate_name_secure` is given a `SEC_NULL_SID` as the `obj_sid`, then its behavior is essentially the same as `mach_port_allocate_name`.

**PARAMETERS**

- **task**
  [in task port] The task acquiring the port right.

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

`mach_port_allocate_name` creates a port. The new port is not a member of any port set. It doesn’t have any extant send or send-once rights. Its make-send count is zero, its sequence number is zero, its queue limit is `MACH_PORT_QLIMIT_DEFAULT`, and it has no queued messages. `name` denotes the receive right for the new port.

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.
Port Manipulation Interface

RELATED INFORMATION
Functions: mach_port_allocate, mach_port_allocate_secure, mach_port_deallocate, mach_port_rename, SSI_context_to_mid, make_sid.
mach_port_deallocate

Function — Releases a user reference for a right

SYNOPSIS

kern_return_t mach_port_deallocate
    (mach_port_t task,
     mach_port_t name);

DESCRIPTION

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

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

SECURITY

The requesting task must hold tsv_remove_name permission to the task port task.

PARAMETERS

    task
        [in task port] 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.
mach_port_destroy

Function — Removes a task’s rights for a name

SYNOPSIS

kern_return_t mach_port_destroy
  (mach_port_t task;
   mach_port_t name);

DESCRIPTION

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

For most purposes, mach_port_mod_refs and mach_port_deallocate are preferable.

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

If name denotes a receive right that is a member of a port set, the receive right is implicitly removed from the port set. If there is a port-destroyed request registered for the port, then the receive right is not actually destroyed, but instead is sent in a port-destroyed notification. (Note: Port destroyed notifications are currently planned for deletion.) If there is no registered port-destroyed request, remaining messages queued to the port are destroyed and extant send and send-once rights turn into dead names. If those send and send-once rights have dead-name requests registered, then dead-name notifications are generated for them.

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

If name denotes a send-once, send, and/or receive right, and it has a dead-name request registered, then the registered send-once right is used to produce a port-deleted notification for the name.

SECURITY

The requesting task must hold tsv_remove_name permission to the task port task.

PARAMETERS

  task

[in task port] 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.
mach_port_extract_right

Function — Extracts a port right from a task

SYNOPSIS

```c
kern_return_t mach_port_extract_right
    (mach_port_t task,
     mach_port_t name,
     mach_msg_type_name_t desired_type,
     mach_port_t* right,
     mach_msg_type_name_t* acquired_type);
```

DESCRIPTION

The `mach_port_extract_right` function extracts a port right from the target task and returns it to the caller as if the task sent the right voluntarily, using `desired_type` as the value of `msgt_name`. See `mach_msg`.

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

SECURITY

The requesting task must hold `tsv_extract_right` permission to the task port `task`. The requesting task must also have permission to hold the port right extracted.

PARAMETERS

- **task**
  [in task port] 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 port] 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.
**mach_port_get_receive_status**

**Function** — Returns the status of a receive right

**SYNOPSIS**

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

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `tsv_observe_pns_info` permission to the task port `task`.

**PARAMETERS**

- **task**
  [in task port] The task holding the receive right.

- **name**
  [in scalar] The task’s name for the receive right.

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

**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.
mach_port_get_receive_status

RELATED INFORMATION

Functions:  
mach_port_set qlimit,  
mach_port_set_mscnt,  
mach_port_set_seqno.

Data Structures: mach_port_status.
mach_port_get_refs

Function — Retrieves the number of user references for a right

SYNOPSIS

kern_return_t mach_port_get_refs
    (mach_port_t task,
     mach_port_t name,
     mach_port_right_t right,
     mach_port_urefs_t* refs);

DESCRIPTION

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

If name denotes a right, but not the type of right specified, then zero is returned. Otherwise a positive number of user references is returned. Note a name may simultaneously denote send and receive rights.

SECURITY

The requesting task must hold tsv_observe_pns_info permission to the task port task.

PARAMETERS

    task
        [in task port] 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.
mach_port_get_refs

RETURN VALUE

KERN_INVALID_NAME

name did not denote a right.

RELATED INFORMATION

Functions: mach_port_mod_refs.
mach_port_get_set_status

Function — Returns the members of a port set

SYNOPSIS

```c
kern_return_t mach_port_get_set_status
    (mach_port_t task,
     mach_port_t name,
     mach_port_array_t* members,
     mach_msg_type_number_t* count);
```

DESCRIPTION

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

SECURITY

The requesting task must hold `tsv_observe_pns_info` permission to the task port `task`.

PARAMETERS

```c
task
    [in task port] The task holding the port set.

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

members
    [out pointer to dynamic array of `mach_port_t`] The task’s names for the port set’s members.

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

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 port set.

RELATED INFORMATION

Functions: mach_port_move_member, vm_deallocate.
**mach_port_insert_right**

**Function** — Inserts a port right into a task

**SYNOPSIS**

```c
kern_return_t mach_port_insert_right(
    mach_port_t task,
    mach_port_t name,
    mach_port_t right,
    mach_msg_type_name_t right_type);
```

**DESCRIPTION**

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

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

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

If *right_type* is MACH_MSG_TYPE_MAKE_SEND, MACH_MSG_TYPE_MOVE_SEND, or MACH_MSG_TYPE_COPY_SEND, then a send right is inserted. If the target already holds send or receive rights for the port, then *name* should denote those rights in the target. Otherwise, *name* should be unused in the target. If the target already has send rights, then those send rights gain an additional user reference. Otherwise, the target gains a send right, with a user reference count of one.

If *right_type* is MACH_MSG_TYPE_MAKE_SEND_ONCE or MACH_MSG_TYPE_MOVE_SEND_ONCE, then a send-once right is inserted. The *name* should be unused in the target. The target gains a send-once right.

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

**SECURITY**

The requesting task must hold *tsv_add_name* permission to the task port *task*. The task having *task* as it’s task port must also hold the appropriate *av_hold_receive*, *av_hold_send* or *av_hold_send_once* permission to the port associated with *name*. 
mach_port_insert_right

PARAMETERS

\textit{task}

[\textit{in task port}] The task which gets the caller's right.

\textit{name}

[\textit{in scalar}] The name by which \textit{task} will know the right.

\textit{right}

[\textit{in random port}] The port right.

\textit{right_type}

[\textit{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

\textit{name} already denoted a right.

KERN_INVALID_CAPABILITY

\textit{right} was null or dead.

KERN_UREFS_OVERFLOW

Inserting the right would overflow \textit{name}'s user-reference count.

KERN_RIGHT_EXISTS

\textit{task} already had rights for the port, with a different name.

RELATED INFORMATION

Functions: \texttt{mach_port_extract_right}, \texttt{mach_msg}.
**mach_port_mod_refs**

**Function** — Changes the number of user refs for a right

**SYNOPSIS**

```c
kern_return_t mach_port_mod_refs
    (mach_port_t task,                 task,
     mach_port_t name,                 name,
     mach_port_right_t right,          right,
     mach_port_delta_t delta);         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.

**SECURITY**

If the port is destroyed as a result of this request, the requesting task must hold *tsv_remove_name* permission to the task port *task*.

**PARAMETERS**

*task*  
[in task port] 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_INVALID_VALUE  
The user-reference count would become negative.

KERN_UREFS_OVERFLOW  
The user-reference count would overflow.

RELATED INFORMATION

Functions: mach_port_destroy, mach_port_get.refs.
mach_port_move_member

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

SYNOPSIS

```c
kern_return_t mach_port_move_member
    (mach_port_t task,               /* task holding port set and receive */
     mach_port_t member,            /* task's name for receive right */
     mach_port_t after);         /* task's name for port set */
```

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.

SECURITY

The requesting task must hold `tsv_manipulate_port_set` permission to the task `task`.

PARAMETERS

- **task**
  [in task port] 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.
mach_port_move_member

KERN_NOT_IN_SET

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

RELATED INFORMATION

Functions: `mach_port_get_set_status`, `mach_port_get_receive_status`. 
mach_port_names

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

SYNOPSIS

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

DESCRIPTION

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

SECURITY

The requesting task must hold `tsv_observe_pns_info` permission to the task port `task`.

PARAMETERS

- **task**
  [in task port] The task whose port name space is queried.

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

- **ncount**
  [out scalar] The number of names returned.

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

- **tcount**
  [out scalar] Should be the same as `ncount`. 
NOTES
This interface is machine word length specific because of the port name parameter.

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: `mach_port_type`, `vm_deallocate`. 
mach_port_rename

**Function** — Change a task’s name for a right

**SYNOPSIS**

```c
kern_return_t mach_port_rename
    (mach_port_t task,
     mach_port_t old_name,
     mach_port_t new_name);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `tsv_port_rename` permission to the task port `task`.

**PARAMETERS**

- `task`
  - [in task port] 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.
mach_port_rename

RELATED INFORMATION

Functions: mach_port_names.
mach_port_request_notification

Function — Request a notification of a port event

SYNOPSIS

```c
kern_return_t mach_port_request_notification
    (mach_port_t task,
     mach_port_t name,
     mach_msg_id_t variant,
     mach_port_mscount_t sync,
     mach_port_t notify,
     mach_msg_type_name_t notify_type,
     mach_port_t* previous);
```

DESCRIPTION

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

The `variant` argument takes the following values:

MACH_NOTIFY_PORT_DESTROYED

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

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

MACH_NOTIFY_DEAD_NAME

The call requests a dead-name notification. `name` specifies send, receive, or send-once rights for a port. If the port is destroyed (and the right remains, becoming a dead name), then a dead-name notification which carries the name of the right will be sent to the registered send-once right. If `sync` is non-zero, the `name` may specify a dead name, and a dead-name notification is immediately generated.

Whenever a dead-name notification is generated, the user reference count of the dead name is incremented. For example, a send right with two user refs has a registered dead-name request. If the port is destroyed, the send right turns into a dead name with three user refs (instead of two), and a dead-name notification is generated.
If the name is made available for reuse, perhaps because of `mach_port_destroy` or `mach_port_mod_refs`, or the name denotes a send-once right which has a message sent to it, then the registered send-once right is used to generate a port-deleted notification instead.

**MACH_NOTIFY_NO_SENDERS**

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

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

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

**SECURITY**

The requesting task must hold `tsv_register_notification` permission to the task port `task`.

**PARAMETERS**

- `task`<br>[in task port] The task holding the specified right.
- `name`<br>[in scalar] The task’s name for the right.
- `variant`<br>[in scalar] The type of notification.
- `sync`<br>[in scalar] Some variants use this value to overcome race conditions.
- `notify`<br>[in notify port] A send-once right, to which the notification will be sent.
Port Manipulation Interface

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

previous
[out notify port] 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.

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_port_get_receive_status.
**mach_port_set_mscount**

*Function* — Changes the make-send count of a port

**SYNOPSIS**

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

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `tsv_alter_pns_info` permission to the task port `task`.

**PARAMETERS**

- `task`  
  [in task port] 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.
Port Manipulation Interface

RELATED INFORMATION

Functions: mach_port_get_receive_status, mach_port_set qlimit.
mach_port_set qlimit

Function — Changes the queue limit of a port

SYNOPSIS

kern_return_t mach_port_set qlimit
    (mach_port_t task,
     mach_port_t name,
     mach_port_msgcount_t qlimit);

DESCRIPTION

The mach_port_set qlimit function changes the queue limit of task’s receive right named name. Valid values for qlimit are between zero and MACH_PORT_QLIMIT_MAX (defined in mach.h), inclusive.

SECURITY

The requesting task must hold tsv alters_pns_info permission to the task port task.

PARAMETERS

  task
    [in task port] The task owning the receive right.
  name
    [in scalar] task’s name for the receive right.
  qlimit
    [in scalar] The number of messages which may be queued to this port without causing the sender to block.

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_receive_status`, `mach_port_set_mscout`. 
mach_port_set_seqno

Function — Changes the sequence number of a port

SYNOPSIS

    kern_return_t mach_port_set_seqno
        (mach_port_t task,
         mach_port_t name,
         mach_port_seqno_t seqno);

DESCRIPTION

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

SECURITY

The requesting task must hold tsv_alter_pns_info permission to the task port task.

PARAMETERS

    task
        [in task port] 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_receive_status
**mach_port_type/mach_port_type_secure**

Function — Return information about a task’s port name

**SYNOPSIS**

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

kern_return_t mach_port_type_secure
    (mach_port_t task,
     mach_port_t name,
     mach_port_type_t* ptype,
     security_id_t* obj_sid,
     mach_access_vector_t av);
```

**DESCRIPTION**

The `mach_port_type` function returns information about `task`'s rights for a specific name in its port name space. The `mach_port_type_secure` function returns information about `task`’s rights, `task`’s access, and the security id for the port associated with a specific name in its port name space. The returned `ptype` is a bit-mask indicating what rights `task` holds with this name. The bit-mask is composed of the following bits:

- **MACH_PORT_TYPE_SEND**
  The name denotes a send right.

- **MACH_PORT_TYPE_RECEIVE**
  The name denotes a receive right.

- **MACH_PORT_TYPE_SEND_ONCE**
  The name denotes a send-once right.

- **MACH_PORT_TYPE_PORT_SET**
  The name denotes a port set.

- **MACH_PORT_TYPE_DEAD_NAME**
  The name is a dead name.

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

- **MACH_PORT_TYPE_MAREQUEST**
  A msg-accepted request for the right is pending. (Note: This feature is planned for deletion.)
MACH_PORT_TYPE_COMPAT
The port right was created in the compatibility mode.

SECURITY
The requesting task must hold tsv_observe_pns_info permissions to the task port task.

PARAMETERS

  task
  [in task port] 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.

  obj_sid
  [out security id] The security identifier of the port associated with the port right. SEC_NULL_SID if name is a port set or a dead name.

  av
  [out access vector] The access vector indicating task’s allowed accesses to 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_receive_status, mach_port_get_set_status.
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.

SECURITY
The requesting task must hold tsv_add_name permission to its own task port.

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 port]
Any other value indicates success.
Port Manipulation Interface

RELATED INFORMATION

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

**Function** — Allocates a region of virtual memory

**SYNOPSIS**

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

kern_return_t vm_allocate_secure
    (mach_port_t target_task,
     vm_address_t* address,
     vm_size_t size,
     boolean_t anywhere,
     security_id_t obj_sid);
```

**DESCRIPTION**

The `vm_allocate` and `vm_allocate_secure` functions allocate a region of virtual memory in the specified task’s address space. A new region is always zero filled. The physical memory is not allocated until an executing thread references the new virtual memory. In addition to allocating a region of virtual memory, `vm_allocate_secure` associates a specific object security identifier with the memory region.

If `anywhere` is true, 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 `vm_statistics` to find the current virtual page size.

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

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

**SECURITY**

The requesting task must hold `tsv_allocate_vm_region` permission to `target_task` and `mosv_map_vm_region` to the object port of the memory object backing the region for the `address` specified. Permissions to the memory are determined by the permissions that `target_task` has to the memory object associated with the allocated memory.
If no object security identifier is provided, the memory is allocated with an object security identifier derived from target_task's subject security identifier.

PARAMETERS

\begin{itemize}
\item \textit{target_task} \hspace{1cm} [in task port] The port for the task in whose address space the region is to be allocated.
\item \textit{address} \hspace{1cm} [pointer to in/out scalar] The starting address for the region. If there is not enough room following the address, the kernel does not allocate the region. The kernel returns the starting address actually used for the allocated region.
\item \textit{size} \hspace{1cm} [in scalar] The number of bytes to allocate.
\item \textit{anywhere} \hspace{1cm} [in scalar] Placement indicator. If false, the kernel allocates the region starting at \textit{address}. If true, the kernel allocates the region wherever enough space is available within the address space. The kernel returns the starting address actually used in \textit{address}.
\item \textit{obj_sid} \hspace{1cm} [in security id] The security identifier to be associated with the region to be allocated.
\end{itemize}

NOTES

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

To establish different protections for the new region, use the \texttt{vm_protect} and \texttt{vm_inherit} functions.

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

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

RETURN VALUE

KERN_INVALID_ADDRESS
The specified address is illegal.

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

RELATED INFORMATION
Functions: task_get_special_port, vm_deallocate, vm_inherit, vm_protect, vm_region, vm_statistics.
vm_copy

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

SYNOPSIS

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

DESCRIPTION

The `vm_copy` function copies a source region to a destination region within 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.

SECURITY

The requesting task must hold `tsv_copy_vm` permission to `target_task`.

In the current implementation the data copied retains the security identifier assigned to the memory region from which the data came.

PARAMETERS

- **target_task**
  [in task port] 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, vm_statistics.
vm_deallocate

Function — De-allocates a region of virtual memory

SYNOPSIS

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

DESCRIPTION

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

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

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

SECURITY

The requesting task must hold tsv_deallocate_vm_region permission to
target_task.

PARAMETERS

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

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

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

NOTES

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

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_statistics.
vm_inherit

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

**SYNOPSIS**

```c
kern_return_t vm_inherit(
    mach_port_t target_task,
    vm_address_t address,
    vm_size_t size,
    vm_inherit_t new_inheritance);
```

**DESCRIPTION**

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

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

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

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

**SECURITY**

The requesting task must hold `tsv_set_vm_region_inherit` permission to `target_task`.

**PARAMETERS**

- `target_task` [in task port] The port for the task whose address space contains the region.
- `address` [in scalar] The starting address for the region.
Virtual Memory Interface

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, norma_task_clone.
vm_machine_attribute

Function — Sets and gets special attributes of a memory region

SYNOPSIS

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

DESCRIPTION

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

SECURITY

The requesting task must hold tsv_access_machine_attribute permission to target_task.

PARAMETERS

target_task
    [in task port] 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
MATTR_MIGRATE
  Migratability

MATTR_REPLICATE
  Replicability

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

MATTR_VAL_OFF
  (generic) turn attribute off

MATTR_VAL_ON
  (generic) turn attribute on

MATTR_VAL_GET
  (generic) return current value

MATTR_VAL_CACHE_FLUSH
  flush from all caches

MATTR_VAL_DCACHE_FLUSH
  flush from data caches

MATTR_VAL_ICACHE_FLUSH
  flush from instruction caches

\textbf{NOTES}

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

\textbf{RETURN VALUE}

\texttt{KERN_INVALID_ADDRESS}
  The address is illegal or specifies a non-allocated region.

\textbf{RELATED INFORMATION}

Functions: \texttt{vm\_wire}. 
vm_map

**Function** — Maps a memory object to a task’s address space

**SYNOPSIS**

```c
kern_return_t vm_map(mach_port_t target_task,
                        vm_address_t* address,
                        vm_size_t size,
                        vm_address_t mask,
                        boolean_t anywhere,
                        mach_port_t memory_object,
                        vm_offset_t offset,
                        boolean_t copy,
                        vm_prot_t cur_protection,
                        vm_prot_t max_protection,
                        vm_inherit_t inheritance);
```

**DESCRIPTION**

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

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

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

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

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

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

SECURITY

The requesting task must hold \texttt{tsv\_allocate\_vm\_region} permission to \texttt{target\_task} and \texttt{mosv\_map\_vm\_region} to the memory objects object port. \texttt{target\_task}'s access to the mapped memory is determined by its permission to \texttt{memory\_object}.

PARAMETERS

\texttt{target\_task}
\hspace{1cm} [in task port] The port for the task to whose address space the memory object is to be mapped.

\texttt{address}
\hspace{1cm} [pointer to in/out scalar] The starting address for the mapped object. The mapped object will start at the beginning of the page containing \texttt{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.

\texttt{size}
\hspace{1cm} [in scalar] The number of bytes to allocate for the object. The kernel rounds this number up to an integral number of virtual pages.

\texttt{mask}
\hspace{1cm} [in scalar] Alignment restrictions for starting address. Bits turned on in the mask will not be turned on in the starting address.

\texttt{anywhere}
\hspace{1cm} [in scalar] Placement indicator. If false, the kernel allocates the object's region starting at \texttt{address}. If true, the kernel allocates the region anywhere at or following \texttt{address} that there is enough space available within the address space. The kernel returns the starting address actually used in \texttt{address}.

\texttt{memory\_object}
\hspace{1cm} [in abstract-memory-object port] The port naming the abstract memory object. If \texttt{MEMORY\_OBJECT\_NULL} is specified, the kernel allocates zero-filled memory, as with \texttt{vm\_allocate}.

\texttt{offset}
\hspace{1cm} [in scalar] An offset within the memory object, in bytes. The kernel maps \texttt{address} to the specified offset.
copy
[in scalar] Copy indicator. If true, the kernel copies the region for the memory object to the specified task’s address space. If false, the region is mapped read-write.

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

VM_PROT_READ
Allows read access.

VM_PROT_WRITE
Allows write access.

VM_PROT_EXECUTE
Allows execute access.

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

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

VM_INHERIT_SHARE
Allows child tasks to share the region.

VM_INHERIT_COPY
Gives child tasks a copy of the region.

VM_INHERIT_NONE
Provides no access to the region for child tasks.

NOTES

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

RELATED INFORMATION

Functions: memory_object_init, vm_allocate.
**vm_protect**

**Function** — Sets access privileges for a region of virtual memory

**SYNOPSIS**

```c
kern_return_t vm_protect(mach_port_t target_task, vm_address_t address, vm_size_t size, boolean_t set_maximum, vm_prot_t new_protection);
```

**DESCRIPTION**

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

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

The enforcement of virtual memory protection is machine-dependent. Nominal-ly, 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.

**SECURITY**

The requesting task must hold `tsv_chg_vm_region_prot` permission to `target_task`.

**PARAMETERS**

- **target_task**
  
  [in task port] The port for the task whose address space contains the region.

- **address**
  
  [in scalar] The starting address for the region.
Virtual Memory Interface

\[ size \]
[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: \texttt{vm_inherit}, \texttt{vm_region}. 
vm_read

Function — Reads a task’s virtual memory

SYNOPSIS

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

DESCRIPTION

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

SECURITY

The requesting task must hold `tsv_read_vm_region` permission to `target_task`.

In the current implementation the data read retains the security identifier assigned to the memory region from which the data came.

PARAMETERS

- **target_task**
  [in task port] 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] The array of data returned by the read.

- **data_count**
  [out scalar] The number of bytes in the returned array. The count converts to an integral number of pages.
Virtual Memory Interface

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

RETURN VALUE

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

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

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

RELATED INFORMATION
Functions: vm_copy, vm_deallocate, vm_write.
vm_region/vm_region_secure

**Function** — Returns information on a region of virtual memory

**SYNOPSIS**

```c
kern_return_t vm_region(mach_port_t target_task,
                         vm_address_t* address,
                         vm_size_t* size,
                         vm_prot_t* mach_protection,
                         vm_prot_t* max_protection,
                         vm_inherit_t* inheritance,
                         boolean_t* shared,
                         mach_port_t* object_name,
                         vm_offset_t* offset);
```

```c
kern_return_t vm_region_secure(mach_port_t target_task,
                                vm_address_t* address,
                                vm_size_t* size,
                                vm_prot_t* mach_protection,
                                vm_prot_t* protection,
                                vm_prot_t* max_protection,
                                vm_inherit_t* inheritance,
                                boolean_t* shared,
                                mach_port_t* object_name,
                                vm_offset_t* offset,
                                security_id_t* obj_sid,
                                mach_access_vector_t av);
```

**DESCRIPTION**

The `vm_region` and `vm_region_secure` functions return information on a region within the specified task’s address space. `vm_region_secure` also returns protection, which incorporates `mach_protection` and the memory protections from the access vector; `obj_sid`, the security identifier for the region containing the address; and `av`, `target_task`'s access vector for the region.

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`.

**SECURITY**

The requesting task must hold `tsv_get_vm_region_info` permission to `target_task`. 

---

Mach 3 Kernel Interfaces 101
PARAMETERS

target_task
[in task port] 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.

mach_protection
[out scalar] The current Mach protection for the region (i.e., the original protection value).

protection
[out scalar] The current protection for the region which incorporates mach_protection and the memory protections from the access vector, av.

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

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

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

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

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

obj_sid
[out security id] The security identifier for the memory object associated with the memory region containing address.

av
[out access vector] The access vector indicating target_task’s allowed access to the region containing address.
vm_region/vm_region_secure

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

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

SYNOPSIS

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

DESCRIPTION

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

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

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

SECURITY

The requesting task must hold `tsv_get_vm_statistics` permission to `target_task`.

PARAMETERS

`target_task`
[in task port] The task that is requesting the statistics.

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: `task_info`.

Data Structures: `vm_statistics`.
vm_wire

**Function** — Specifies the pageability of a region of virtual memory

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

```c
kern_return_t vm_wire
    (mach_port_t host_priv,           host_priv,
     mach_port_t target_task,      target_task,
     vm_address_t address,           address,
     vm_size_t size,                size,
     vm_prot_t wired_access);      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 page is wired into physical memory if any task accessing it has a non-null `wired_access` value 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 `vm_statistics` to find the current virtual page size.

**SECURITY**

The requesting task must hold `tsv_wire_vm_for_task` permission to `target_task` and `hpsv_wire_vm` permission to `host_priv`.

**PARAMETERS**

- `host_priv` [in host-control port] The host control port for the host on which `target_task` executes.
- `target_task` [in task port] The port for the task whose address space contains the region.
- `address` [in scalar] The starting address for the region.
Virtual Memory Interface

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 call requires the privileged host port on which target_task executes because of the privileged nature of committing physical memory.

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

RETURN VALUE

KERN_INVALID_HOST
The privileged host port was not specified

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

RELATED INFORMATION
Functions: thread_wire.
vm_write

Function — Writes data to a task’s virtual memory

SYNOPSIS

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

SECURITY

The requesting task must hold tsv_write_vm_region permission to target_task.

The SID of memory region to which the data is written, is unaffected. If the write results in creation of a new memory region, the SID assigned to that region will be the default memory object sid for the target_task.

PARAMETERS

target_task
[in task port] 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
[in pointer to page aligned array of bytes] An array of data to be written.

data_count
[in scalar] The number of bytes in the array. The size of the array must convert to an integral number of pages.
NOTES
This interface is machine word length specific because of the virtual address parameter.

RETURN VALUE

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

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

RELATED INFORMATION
Functions: vm_copy, vm_protect, vm_read, vm_statistics.
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.
memory_object_change_attributes

**Function** — Changes various performance related attributes

**SYNOPSIS**

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

**DESCRIPTION**

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

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

**SECURITY**

The requesting task must hold `mcsv_set_attributes` permission to `memory_control`.

**PARAMETERS**

- `memory_control` [in memory-cache-control port] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a `memory_object_init` call.
- `may_cache_object` [in scalar] Cache indicator. If true, the kernel can cache data associated with the memory object, even if virtual memory references to it are removed.
- `copy_strategy` [in scalar] How the kernel should handle copying of regions associated with the memory object. Valid values are:
  - `MEMORY_OBJECT_COPY_NONE` Use normal procedure when copying the memory object’s data. Normally, the kernel requests each page with read access, copies the data, and then (optionally) flushes the data.
MEMORY_OBJECT_COPY_CALL
Notify the memory manager (via memory_object_copy) before copying any data.

MEMORY_OBJECT_COPY_DELAY
Use copy-on-write technique. This strategy allows the kernel to efficiently copy large amounts of data and guarantees that the memory manager will not externally modify the data. It is the most commonly used copy strategy.

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

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

NOTES
Sharing cached data among all the clients of a memory object can have a major impact on performance, especially if it can be extended across successive, as well as concurrent, uses. For example, the memory objects that represent program images can be used regularly by different programs. By retaining the data for these memory objects in cache, the number of secondary storage accesses can be reduced significantly.

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

RELATED INFORMATION
Functions: memory_object_change_completed, memory_object_copy, memory_object_get_attributes, memory_object_ready, memory_object_set_attributes (old form).
memory_object_change_completed

Server Interface — Indicates completion of an attribute change call

LIBRARY

Not declared anywhere.

SYNOPSIS

```c
kern_return_t memory_object_change_completed
    (mach_port_t reply_port,
    boolean_t may_cache_object,
    memory_object_copy_strategy_t copy_strategy);
```

```c
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,
    boolean_t may_cache_object,
    memory_object_copy_strategy_t copy_strategy);
```

DESCRIPTION

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

When the kernel completes the requested changes, it calls memory_object_change_completed (asynchronously) using the port explicitly provided in the memory_object_change_attributes call. A response is generated so that the manager can synchronize with changes to the copy strategy (which affects the manner in which pages will be requested) and a termination message possibly resulting from un-cacheing a not-mapped object.

PARAMETERS

`reply_port`

[in reply port] The port named 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.

`may_cache_object`

memory_object_change_completed

\textit{copy\_strategy}
\begin{itemize}
\item [in scalar] The new copy strategy.
\end{itemize}

\textbf{NOTES}

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

\textbf{RETURN VALUE}

Irrelevant.

\textbf{RELATED INFORMATION}

Functions: \texttt{memory\_object\_change\_attributes}, \texttt{memory\_object\_server}, \texttt{seqnos\_memory\_object\_server}.
External Memory Management Interface

memory_object_copy

Server Interface — Indicates that a memory object has been copied

LIBRARY

Not declared anywhere.

SYNOPSIS

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

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

DESCRIPTION

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

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

The kernel makes the memory_object_copy call only if:

- The memory manager had previously set the old object’s copy strategy attribute to MEMORY_OBJECT_COPY_CALL (using memory_object_change_attributes or memory_object_ready).
- A user of the old object has asked the kernel to copy it.
Cached pages from the old memory object at the time of the copy are handled as follows:

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

The memory manager should treat the new memory object as temporary. In other words, the memory manager should not change the new object’s contents or allow it to be mapped in another client. The memory manager can use the `memory_object_data_unavailable` call to indicate that the appropriate pages of the old object can be used to fulfill a data request.

**PARAMETERS**

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

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

- `old_memory_control`[in memory-cache-control port] The kernel memory cache control port for the old memory object.

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

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

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

**NOTES**

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

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause `mach_msg_server` to remove the old and new memory cache control port references.

RELATED INFORMATION

Functions:

- `memory_object_change_attributes`
- `memory_object_data_unavailable`
- `memory_object_init`
- `memory_object_ready`
- `memory_object_server`
- `seqnos_memory_object_server`
memory_object_data_error

Function — Indicates no data for a memory object

SYNOPSIS

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

DESCRIPTION

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

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

SECURITY

The requesting task must hold `mcsv_provide_data` permission to `memory_control`.

PARAMETERS

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

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

- `size` [in scalar] The number of bytes of data (starting at `offset`). The number must convert to an integral number of memory object pages.
**External Memory Management Interface**

\[\text{reason}\]

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

**NOTES**

If \textit{reason} has a system code of \texttt{err_kern}, the kernel will substitute an error value of \texttt{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: \texttt{memory_object_data_request}, \texttt{memory_object_data_supply}, \texttt{memory_object_data_unavailable}. 

**memory_object_data_provided**

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

**SYNOPSIS**

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

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `mcsv_provide_data` permission to `memory_control`.

**PARAMETERS**

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

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

- `data`  
  [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.

- `lock_value`  
  [in scalar] One or more forms of access not permitted for the specified data. Valid values are:
VM_PROT_NONE
   Prohibits no access (that is, all forms of access are permitted).

VM_PROT_READ
   Prohibits read access.

VM_PROT_WRITE
   Prohibits write access.

VM_PROT_EXECUTE
   Prohibits execute access.

VM_PROT_ALL
   Prohibits all forms of access.

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

memory_object_data_provided is the old form of
memory_object_data_supply.

CAUTIONS
A memory manager must be careful 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_supply, memory_object_data_unavailable,
memory_object_lock_request.
memory_object_data_request

Server Interface — Requests data from a memory object

LIBRARY
Not declared anywhere.

SYNOPSIS

```c
kern_return_t memory_object_data_request
    (mach_port_t memory_object,
     mach_port_t memory_control,
     vm_offset_t offset,
     vm_size_t length,
     vm_prot_t desired_access);
```

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

DESCRIPTION

A memory_object_data_request function is called as the result of a kernel message requesting data from the specified memory object, for at least the access specified.

The kernel issues this call after a cache miss (that is, a page fault for which the kernel does not have the data). The kernel requests only amounts of data that are multiples of the page size included in the memory_object_init call.

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

memory_object

[in abstract-memory-object port] The abstract memory object port that represents the memory object data, as supplied to the kernel in a `vm_map` call.

seqno

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

memory_control

[in memory-cache-control port] 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_datasupply`, `memory_object_data_unavailable`, `memory_object_server`, `seqnos_memory_object_server`. 
memory_object_data_return

Server Interface — Writes data back to a memory object

LIBRARY
Not declared anywhere.

SYNOPSIS

```c
kern_return_t memory_object_data_return
    (mach_port_t memory_object,
     mach_port_t memory_control,
     vm_offset_t offset,
     vm_offset_t data,
     vm_size_t data_count,
     boolean_t dirty,
     boolean_t kernel_copy);
```

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

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 port] The abstract memory object port that represents the memory object data, as supplied to the kernel in a `vm_map` call.
**seqno**

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

**memory_control**

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

**offset**

[in scalar] The offset within the memory object.

**data**

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

**data_count**

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

**dirty**

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

**kernel_copy**

[in scalar] If TRUE, the kernel has kept a copy of the page.

**NOTES**

The kernel can flush clean (that is, un-modified) non-precious pages at its own discretion. As a result, the memory manager cannot rely on the kernel to keep a copy of its data or even to provide notification that its data has been discarded.

**RETURN VALUE**

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

**RELATED INFORMATION**

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

*Function* — Supplies data for a region of a memory object

**SYNOPSIS**

```c
kern_return_t memory_object_data_supply(mach_port_t memory_control, vm_offset_t offset, vm_offset_t data, mach_msg_type_number_t data_count, boolean_t deallocate, vm_prot_t lock_value, boolean_t precious, mach_port_t reply_port);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `mcsv_change_page_locks`, `mcsv_make_page_precious` and `mcsv_provide_data` permission to `memory_control`.

**PARAMETERS**

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

- `offset`

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

- `data`

  [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.
External Memory Management Interface

\textit{deallocate}

[in scalar] If TRUE, the pages to be copied (starting at \texttt{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.

\textit{lock_value}

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

- \texttt{VM\_PROT\_NONE}
  Prohibits no access (that is, all forms of access are permitted).

- \texttt{VM\_PROT\_READ}
  Prohibits read access.

- \texttt{VM\_PROT\_WRITE}
  Prohibits write access.

- \texttt{VM\_PROT\_EXECUTE}
  Prohibits execute access.

- \texttt{VM\_PROT\_ALL}
  Prohibits all forms of access.

\textit{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.

\textit{reply_port}

[in reply port] A port to which the kernel should send a \texttt{memory\_object\_supply\_completed} to indicate the status of the accepted data. \texttt{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_data_supply

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_provided (old form), memory_object_data_request, memory_object_data_unavailable, memory_object_lock_request, memory_object_supply_completed.
memory_object_data_unavailable

Function — Indicates no data for a memory object

SYNOPSIS

kern_return_t memory_object_data_unavailable
    (mach_port_t memory_control, vm_offset_t offset, vm_size_t size);

DESCRIPTION

The memory_object_data_unavailable function indicates that the memory manager cannot provide the kernel with the data requested for the given region. Instead, the kernel should provide the data for this region.

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

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

• When the object was created by a memory_object_copy. In this case, the kernel should copy the region from the original memory object.

• When the object is a normal user-created memory object. In this case, the kernel should provide unlocked zero-filled pages for the region.

SECURITY

The requesting task must hold mcsv_provide_data permission to memory_control.

PARAMETERS

memory_control
    [in memory-cache-control port] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a memory_object_init or a memory_object_create call.

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

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_copy, memory_object_create, memory_object_data_error, memory_object_data_request, memory_object_data_supply.
memory_object_data_unlock

Server Interface — Requests access to a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t memory_object_data_unlock
    (mach_port_t memory_object,
     mach_port_t memory_control,
     vm_offset_t offset,
     vm_size_t length,
     vm_prot_t desired_access);

seqnos_memory_object_data_unlock

Sequence Number form

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

DESCRIPTION

A memory_object_data_unlock function is called as the result of a kernel mes-
sage 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 port] The abstract memory object port that
    represents the memory object data, as supplied to the kernel in a
    vm_map call.

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

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

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

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

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

- **VM_PROT_READ**
  Allows read access.

- **VM_PROT_WRITE**
  Allows write access.

- **VM_PROT_EXECUTE**
  Allows execute access.

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

**RELATED INFORMATION**
Functions: memory_object_lock_completed, memory_object_lock_request, memory_object_server, seqnos_memory_object_server.
memory_object_data_write

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

LIBRARY
Not declared anywhere.

SYNOPSIS

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

```
seqnos_memory_object_data_write
    Sequence Number form

kern_return_t seqnos_memory_object_data_write
    (mach_port_t memory_object, mach_port_t memory_control,
     mach_port_seqno_t seqno, offset, data, data_count);
```

DESCRIPTION

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

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

PARAMETERS

memory_object
[in abstract-memory-object port] The abstract memory object port that represents the memory object data, as supplied to the kernel in a vm_map call.
**memory_object_data_write**

*seqno*

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

*memory_control*

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

*offset*

[in scalar] The offset within the memory object.

*data*

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

*data_count*

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

**NOTES**

The kernel can flush clean (that is, un-modified) pages at its own discretion. As a result, the memory manager cannot rely on the kernel to keep a copy of its data or even to provide notification that its data has been discarded.

**RETURN VALUE**

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_return*, *memory_object_set_attributes*, *vm_deallocate*, *memory_object_server*, *seqnos_memory_object_server*. 
memory_object_destroy

Function — Shuts down a memory object

SYNOPSIS

```c
kern_return_t memory_object_destroy
    (mach_port_t memory_control, 
     kern_return_t reason);
```

DESCRIPTION

The `memory_object_destroy` function tells the kernel to shut down the specified memory object. As a result of this call, the kernel no longer supports paging activity or any memory object calls on the memory object. The kernel issues a `memory_object_terminate` call to pass to the memory manager all rights to the memory object port, the memory control port, and the memory name port.

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

SECURITY

The requesting task must hold `mcsv_destroy_object` permission to `memory_control`.

PARAMETERS

`memory_control`

[in memory-cache-control port] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a `memory_object_init` call.

`reason`

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

NOTES

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

RETURN VALUE

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

Functions: `memory_object_lock_request`, `memory_object_terminate`.  

memory_object_destroy
memory_object_get_attributes

Function — Returns current attributes for a memory object

SYNOPSIS

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

DESCRIPTION

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

SECURITY

The requesting task must hold mcsv_get_attributes permission to memory_control.

PARAMETERS

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

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

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

copy_strategy
    [out scalar] How the kernel should handle copying of regions associated with the memory object. Possible values are:

MEMORY_OBJECT_COPY_NONE
    Use normal procedure when copying the memory object’s data. Normally, the kernel requests each page with read access, copies the data, and then (optionally) flushes the data.
MEMORY_OBJECT_COPY_CALL
Notify the memory manager (via `memory_object_copy`) before copying any data.

MEMORY_OBJECT_COPY_DELAY
Use copy-on-write technique. This strategy allows the kernel to efficiently copy large amounts of data and guarantees that the memory manager will not externally modify the data. It is the most commonly used copy strategy.

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

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: `memory_object_change_attributes`, `memory_object_copy`, `memory_object_ready`. 

memory_object_get_attributes
memory_object_init

Server Interface — Initializes a memory object

LIBRARY

Not declared anywhere.

SYNOPSIS

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

seqnos_memory_object_init

Sequence Number form

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

DESCRIPTION

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

When asked to map a memory object for the first time, the kernel responds by making a memory_object_init call on the abstract memory object. This call is provided as a convenience to the memory manager, to allow it to initialize data structures and prepare to receive other requests.

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

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

The kernel holds send rights for the abstract memory object port, and both send and receive rights for the memory cache control and name ports.
The call also supplies the virtual page size to be used for the memory mapping. The memory manager can use this size to detect mappings that use different data structures at initialization time, or to allocate buffers for use in reading data.

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

**PARAMETERS**

*memory_object*  
[in abstract-memory-object port] The abstract memory object port that represents the memory object data, as supplied to the kernel in a `vm_map` call.

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

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

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

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

**NOTES**

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

**RETURN VALUE**

Any return value other than KERN_SUCCESS or MIG_NO_REPLY will cause `mach_msg_server` to remove the memory cache control and name port references.
RELATED INFORMATION

Functions: memory_object_ready, memory_object_terminate, memory_object_server, seqnos_memory_object_server.
memory_object_lock_completed

Server Interface — Indicates completion of a consistency control call

LIBRARY
Not declared anywhere.

SYNOPSIS

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

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

DESCRIPTION

A `memory_object_lock_completed` function is called as the result of a kernel message confirming the kernel’s action in response to a `memory_object_lock_request` call from the memory manager. The memory manager can use the `memory_object_lock_request` call to:

- Alter access restrictions specified in the `memory_object_data_supply` call or a previous `memory_object_lock_request` call.
- Write back modifications made in memory.
- Invalidate its cached data.

When the kernel completes the requested actions, it calls `memory_object_lock_completed` (asynchronously) using the port explicitly provided in the `memory_object_lock_request` call. Because the memory manager cannot know which pages have been modified, or even which pages remain in the cache, it cannot know how many pages will be written back in response to a `memory_object_lock_request` call. Receiving the `memory_object_lock_completed` call is the only sure means of detecting completion. The completion call includes the offset and length values from the consistency request to distinguish it from other consistency requests.
PARAMETERS

reply_port
[in reply] The port named 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_request message.

memory_control
[in memory-cache-control port] 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

Function — Restricts access to memory object data

SYNOPSIS

```c
kern_return_t memory_object_lock_request(
mach_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_port` port.

SECURITY

The requesting task must hold `mcsv_remove_page`, `mcsv_change_page_locks`, and `mcsv_invoke_lock_request` permission to `memory_control`.

PARAMETERS

`memory_control`

[in memory-cache-control port] The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a `memory_object_init` call.
**External Memory Management Interface**

**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
Allows all forms of access.

reply_port
[in reply port] 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_ready**

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

**SYNOPSIS**

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

**DESCRIPTION**

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

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

**SECURITY**

The requesting task must hold `mcsv_set_attributes` permission to `memory_control`.

**PARAMETERS**

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

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

- `copy_strategy` [in scalar] How the kernel should handle copying of regions associated with the memory object. Valid values are:

  - `MEMORY_OBJECT_COPY_NONE`
  
  Use normal procedure when copying the memory object’s data. Normally, the kernel requests each page with read access, copies the data, and then (optionally) flushes the data.
MEMORY_OBJECT_COPY_CALL
Notify the memory manager (via `memory_object_copy`) before copying any data.

MEMORY_OBJECT_COPY_DELAY
Use copy-on-write technique. This strategy allows the kernel to efficiently copy large amounts of data and guarantees that the memory manager will not externally modify the data. It is the most commonly used copy strategy.

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

NOTES
Sharing cached data among all the clients of a memory object can have a major impact on performance, especially if it can be extended across successive, as well as concurrent, uses. For example, the memory objects that represent program images can be used regularly by different programs. By retaining the data for these memory objects in cache, the number of secondary storage accesses can be reduced significantly.

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

RELATED INFORMATION
Functions: `memory_object_change_attributes`, `memory_object_copy`, `memory_object_get_attributes`, `memory_object_init`, `memory_object_set_attributes` (old form).
memory_object_set_attributes

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

**SYNOPSIS**

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

**DESCRIPTION**

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

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

**SECURITY**

The requesting task must hold `mcsv_set_attributes` permission to `memory_control`.

**PARAMETERS**

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

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

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

- `copy_strategy`
  [in scalar] How the kernel should handle copying of regions associated with the memory object. Valid values are:
memory_object_set_attributes

MEMORY_OBJECT_COPY_NONE
Use normal procedure when copying the memory object’s data. Normally, the kernel requests each page with read access, copies the data, and then (optionally) flushes the data.

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

MEMORY_OBJECT_COPY_DELAY
Use copy-on-write technique. This strategy allows the kernel to efficiently copy large amounts of data and guarantees that the memory manager will not externally modify the data. It is the most commonly used copy strategy.

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

NOTES

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

Sharing cached data among all the clients of a memory object can have a major impact on performance, especially if it can be extended across successive, as well as concurrent, uses. For example, the memory objects that represent program images can be used regularly by different programs. By retaining the data for these memory objects in cache, the number of secondary storage accesses can be reduced significantly.

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

Functions: memory_object_change_attributes, memory_object_copy,
memory_object_get_attributes, memory_object_init,
memory_object_ready.
memory_object_supply_completed

**Server Interface** — Indicates completion of a data supply call

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

```c
kern_return_t memory_object_supply_completed
    (mach_port_t reply_port,
     mach_port_t memory_control,
     vm_offset_t offset,
     vm_size_t length,
     kern_return_t result,
     vm_offset_t error_offset);
```

```c
kern_return_t seqnos_memory_object_supply_completed
    (mach_port_t reply_port,
     mach_port_t memory_control,
     vm_offset_t offset,
     vm_size_t length,
     kern_return_t result,
     vm_offset_t error_offset);
```

**DESCRIPTION**

A `memory_object_supply_completed` function is called as the result of a kernel message confirming the kernel’s action in response to a `memory_object_data_supply` call from the memory manager.

When the kernel accepts the pages, it calls `memory_object_supply_completed` (asynchronously) using the port explicitly provided in the `memory_object_data_supply` call. Because the data supply call can provide multiple pages, not all of which the kernel may necessarily accept and some of which the kernel may have to return to the manager (if precious), the kernel provides this response. If the kernel does not accept all of the pages in the data supply message, it will indicate so in the completion response. If the pages not accepted are precious, they will be returned (in `memory_object_data_return` messages) before it sends this completion message. The completion call includes the offset and length values from the supply request to distinguish it from other supply requests.
PARAMETERS

reply_port
[in reply port] The port specified to 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 port] 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_terminate**

Server Interface — Relinquishes access to a memory object

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

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

```c
seqnos_memory_object_terminate
    Sequence Number form

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

**DESCRIPTION**

A `memory_object_terminate` function is called as the result of a kernel message notifying a memory manager that no mappings of the specified memory object remain. The kernel makes this call to allow the memory manager to clean up data structures associated with the de-allocated mappings. The call provides receive rights to the memory cache control and name ports so that the memory manager can destroy the ports (via `mach_port_deallocate`). The kernel also relinquishes its send rights for all three ports.

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

**PARAMETERS**

- **memory_object**
  
  [in abstract-memory-object port] The abstract memory object port that represents the memory object data, as supplied to the kernel in a `vm_map` call.

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

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

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

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

RELATED INFORMATION
Functions: memory_object_destroy, memory_object_init, mach_port_deallocate, memory_object_server, seqnos_memory_object_server.
This chapter discusses the specifics of the kernel’s thread interfaces. This includes status functions related to threads. Properties associated with threads, such as special ports, are included here as well. Functions that apply to more than one thread appear in the task interface chapter.
catch_exception_raise

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

LIBRARY
Not declared anywhere.

SYNOPSIS

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

DESCRIPTION

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

SECURITY

There are no security limitations on this kernel outcall.

PARAMETERS

- `exception_port`[in exception port] The port to which the exception notification was sent.
- `thread`[in thread port] The port to the thread taking the exception.
- `task`[in task port] The port to the task containing the thread taking the exception.
- `exception`[in scalar] The type of the exception, as defined in `<mach/exception.h>`. The machine independent values raised by all implementations are:
EXC_BAD_ACCESS
   Could not access memory. code contains kern_return_t describing error. subcode contains bad memory address.

EXC_BAD_INSTRUCTION
   Instruction failed. Illegal or undefined instruction or operand.

EXC_ARITHMETIC
   Arithmetic exception; exact nature of exception is in code field.

EXC_EMULATION
   Emulation instruction. Emulation support instruction encountered. Details in code and subcode fields.

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

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

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

subcode
   [in scalar] A specific type of code.

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

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

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

- thread_suspend. This keeps the thread from proceeding after the next step.
Thread Interface

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

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

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

**RETURN VALUE**

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

**RELATED INFORMATION**

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

*System Trap* — Returns the thread self port

**LIBRARY**

```c
#include <mach/mach_traps.h>
```

**SYNOPSIS**

```c
mach_port_t mach_thread_self
();
```

**DESCRIPTION**

The `mach_thread_self` function returns send rights to the thread’s own kernel port.

**SECURITY**

The requesting task must hold `thsv_get_thread_kernel_port` permission to its own thread port.

**PARAMETERS**

None

**RETURN VALUE**

[thread-self port] Send rights to the thread’s port.

**RELATED INFORMATION**

Functions: `thread_info`, `thread_set_special_port`. 
Thread Interface

receive_samples

Server Interface — Handles the occurrence of a PC sampling message

LIBRARY

Not declared anywhere.

SYNOPSIS

kern_return_t receive_samples
   (mach_port_t sample_port,
    sample_array_t samples,
    mach_msg_type_number_t samplesCnt);

DESCRIPTION

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

SECURITY

There are no security limitations on this kernel outcall.

PARAMETERS

   sample_port
     [in sample port] The port to which the sample message was sent.

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

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

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

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

System Trap — Attempt a context switch

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

```c
boolean_t swtch()
```

**DESCRIPTION**

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

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

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

**SECURITY**

The requesting task must hold `thsv_can_swtch` permission to its own thread port

**PARAMETERS**

None

**RETURN VALUE**

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

FALSE
The processor has nothing better to do.
```

**RELATED INFORMATION**

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

*System Trap* — Attempt a context switch to low priority

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

```c
boolean_t swtch_pri (int priority);
```

**DESCRIPTION**

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

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

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

**SECURITY**

The requesting task must hold `thsv_can_swtch_pri` permission to its own thread port.

**PARAMETERS**

`priority`

[in scalar] Currently not used.

**RETURN VALUE**

**TRUE**

There are other threads that the processor could run.

**FALSE**

The processor has nothing better to do.
RELATED INFORMATION

Functions: swtch, thread_abort, thread_depress_abort, thread_switch.
thread_abort

Function — Aborts a thread

SYNOPSIS

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

DESCRIPTION

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

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

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

SECURITY

The requesting task must hold `thsv_abort_thread` permission to `target_thread`.

PARAMETERS

target_thread
    [in thread port] The thread to be aborted.

NOTES

By way of comparison, the `thread_suspend` function keeps the target thread from executing any further instructions at the user level, including the return from a system call. The `thread_get_state` function returns the thread’s user state, while `thread_set_state` allows modification of the user state.
A problem occurs if a suspended thread had been executing within a system call. In this case, the thread has, not only a user state, but an associated kernel state. (The kernel state cannot be changed with \texttt{thread_set_state}.) As a result, when the thread resumes, the system call can return, producing a change in the user state and, possibly, user memory.

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

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

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

**CAUTIONS**

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

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

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: \texttt{mach_msg}, \texttt{thread_get_state}, \texttt{thread_info}, \texttt{thread_set_state}, \texttt{thread_suspend}, \texttt{thread_terminate}, \texttt{thread_abort_safely}. 
Thread Interface

thread_create/thread_create_secure

**Function** — Creates a thread within a task

**SYNOPSIS**

```c
kern_return_t thread_create
    (mach_port_t parent_task,
     mach_port_t* child_thread);

kern_return_t thread_create_secure
    (mach_port_t parent_task,
     mach_port_t* child_thread);
```

**DESCRIPTION**

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

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

The `thread_create_secure` function creates a new thread within `parent_task` only if the task had been created by `task_create_secure` and the `parent_task`’s task structure is in an EMPTY state. The state of `child_thread`’s task structure is changed from EMPTY to THREAD_CREATED.

**SECURITY**

For the `thread_create` function, the requesting task must have `tsv_add_thread` permission to `parent_task`. For the `thread_create_secure` function, the requesting task must have `tsv_add_thread_secure` permission to `parent_task`.

**PARAMETERS**

- **parent_task**
  [in task port] The port for the task that is to contain the new thread.

- **child_thread**
  [out thread port] The kernel-assigned name for the new thread.

**NOTES**

To get a new thread running, first use `thread_set_state` to set a processor state for the thread. Then, use `thread_resume` to schedule the thread for execution.
RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: task_create, task_create_secure, task_threads,
thread_get_special_port, thread_get_state, thread_resume,
thread_resume_secure, thread_set_special_port, thread_set_state,
thread_set_state_secure, thread_suspend, thread_terminate.
thread_depress_abort

Function — Cancel thread priority depression

SYNOPSIS

kern_return_t thread_depress_abort
    (mach_port_t thread);

DESCRIPTION

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

SECURITY

The requesting task must hold thsv_abort_thread_depress permission to thread.

PARAMETERS

thread
    [in thread port] Thread whose priority depression is canceled.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

**Function** — Returns a send right to a special port

**SYNOPSIS**

```c
t Kern_return_t thread_get_special_port (mach_port_t thread,
int which_port,
mach_port_t* special_port);
```

**thread_get_exception_port**

**Macro form**

```c
t Kern_return_t thread_get_exception_port (mach_port_t thread,
mach_port_t* special_port)
⇒ thread_get_special_port (thread, THREAD_EXCEPTION_PORT,
special_port)
```

**thread_get_kernel_port**

**Macro form**

```c
t Kern_return_t thread_get_kernel_port (mach_port_t thread,
mach_port_t* special_port)
⇒ thread_get_special_port (thread, THREAD_KERNEL_PORT,
special_port)
```

**DESCRIPTION**

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

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

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

**SECURITY**

The requesting task must hold `thsv_get_thread_exception_port` or `thsv_get_thread_kernel_port` permission to `thread` to get, respectively, the exception port or the kernel port.
PARAMETERS

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

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

THREAD_EXCEPTION_PORT
[exception port] The thread’s exception port. Used to receive exception messages from the kernel.

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Function — Returns the execution state for a thread

SYNOPSIS

kern_return_t thread_get_state
    (mach_port_t target_thread,
     int flavor,
     thread_state_t old_state,
     mach_msg_type_number_t* old_stateCnt);

DESCRIPTION

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

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

SECURITY

The requesting task must hold thsv_get_thread_state permission to target_thread.

PARAMETERS

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

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

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

    old_stateCnt  
        [pointer to in/out scalar] On input, the maximum size of the state array; on output, the returned size of the state array (in units of sizeof (int)). The maximum size is defined by THREAD_STATE_MAX.

RETURN VALUE

    Only generic errors apply.
RELATED INFORMATION

Functions: task_info, thread_info, thread_set_state.
thread_info

**Function** — Returns information about a thread

### SYNOPSIS

```c
kern_return_t thread_info
(mach_port_t target_thread,
 int flavor,
 thread_info_t thread_info,
 mach_msg_type_number_t* thread_infoCnt);
```

### DESCRIPTION

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

### SECURITY

The requesting task must hold `thsv_get_thread_info` permission to `target_thread`.

### PARAMETERS

- **target_thread**
  - [in thread port] The thread for which the information is to be returned.

- **flavor**
  - [in scalar] The type of information to be returned. Valid values are:
    - THREAD_BASIC_INFO
      Returns basic information about the thread, such as the thread’s run state and suspend count. The returned structure is `thread_basic_info` of size `THREAD_BASIC_INFO_COUNT`.
    - THREAD_SCHED_INFO
      Returns scheduling information about the thread, such as priority and scheduling policy. The returned structure is `thread_sched_info` of size `THREAD_SCHED_INFO_SIZE`.

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

- **thread_infoCnt**
  - [pointer to in/out scalar] On input, the size of the info buffer; on output, the returned size of the information structure (in units of `sizeof(int)`). The maximum size is defined by `THREAD_INFO_MAX`. 
RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Data Structures: thread_basic_info, thread_sched_info.
**thread_resume/thread_resume_secure**

Function — Resumes a thread

**SYNOPSIS**

```c
kern_return_t thread_resume
(mach_port_t target_thread);

kern_return_t thread_resume_secure
(mach_port_t target_thread);
```

**DESCRIPTION**

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

The `thread_resume_secure` function decrements the suspend count for `target_thread` by one. The state of `target_thread`'s associated task structure is changed from THREAD_STATE_SET to TASK_READY state.

**SECURITY**

The `thread_resume` function requires that the requesting task hold `thsv_resume_thread` permission to `target_thread`. The `thread_resume_secure` function requires that the requesting task hold `thsv.resume_thread` and `thsv.initiate_secure` permission to `target_thread` and `target_thread`'s associated thread structure must be in the THREAD_STATE_SET state.

**PARAMETERS**

`target_thread`

[in thread port] The thread to be resumed.

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `task_resume`, `task_suspend`, `thread_create`, `thread_create_secure`, `thread_info`, `thread_suspend`, `thread_terminate`. 
**Thread Interface**

**thread_sample**

**Function** — Perform periodic PC sampling for a thread

**SYNOPSIS**

```c
kern_return_t thread_enable_pc_sampling
    (mach_port_t thread, int *ticks; sampled_pc_flavor_t flavor);
kern_return_t thread_disable_pc_sampling
    (mach_port_t thread, int *sample_cnt; sampled_pc_flavor_t flavor);
kern_return_t thread_get_sampled_pcs
    (mach_port_t thread, unsigned *seqno; sampled_pc_t sampled_pcs[], int *sample_cnt);
```

**DESCRIPTION**

These functions cause the program counter (PC) of the specified thread to be sampled periodically (whenever the thread happens to be running at the time of the kernel’s “hardclock” interrupt). The set of PC sample values obtained are saved in buffers.

**SECURITY**

These functions require that the requesting task hold thsv_sample_thread permission to thread.

**PARAMETERS**

`thread`
[in thread port] Thread whose PC is to be sampled.

`ticks`
[out scalar] The kernel’s idea of clock granularity (ticks per second). Don’t trust this.

`flavor`
[in structure] The sampling flavor, which can be any of the following flavors defined in pc_sample.h.

```
SAMPLED_PC_PERIODIC,
SAMPLED_PC_VM_ZFILL_FAULTS,
```
SAMPLED_PC_VM.REACTIVATION_FAULTS,
SAMPLED_PC_VM.PAGIN_FAULTS,
SAMPLED_PC_VM.COM_FAULTS,
SAMPLED_PC_VM.FAULTS.ANY,
SAMPLED_PC_VM.FAULTS.

seqno
[out scalar] The sequence number of the sampled PC’s. This is useful for determining when a collector thread has missed a sample.

sampled_pcs
[out structure] The sampled PCs for thread. A sample contains three fields: a thread-specific unique identifier, a PC value and the type of sample as per flavor.

sample_cnt
[out scalar] The number of sample elements in the kernel for the named task or thread.

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: task_enable_pc_sampling, task_disable_pc_sampling, task_get_sampled_pcs.
thread_set_special_port

**Function** — Sets a special port for a thread

**SYNOPSIS**

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

thread_set_exception_port

**Macro form**

kern_return_t thread_set_exception_port
    (mach_port_t thread,
     mach_port_t special_port)

⇒ thread_set_special_port (thread, THREAD_EXCEPTION_PORT,
                             special_port)

thread_set_kernel_port

**Macro form**

kern_return_t thread_set_kernel_port
    (mach_port_t thread,
     mach_port_t special_port)

⇒ thread_set_special_port (thread, THREAD_KERNEL_PORT, special_port)

**DESCRIPTION**

The thread_set_special_port function sets a special port belonging to thread.

**SECURITY**

The requesting task must hold thsv_set_thread_exception_port or thsv_set_thread_kernel_port permission to thread to set, respectively, the thread’s exception port or kernel port.

**PARAMETERS**

*thread*

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

*which_port*

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

THREAD_EXCEPTION_PORT
[exception port] The thread’s exception port. Used to receive exception messages from the kernel.

THREAD_KERNEL_PORT
[thread-self port] The thread’s kernel port. Used by the kernel to receive messages from the thread.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: mach_thread_self, task_get_special_port, task_set_special_port, thread_create, thread_get_special_port.
Thread Interface

thread_set_state/thread_set_state_secure

**Function** — Sets the execution state for a thread

**SYNOPSIS**

```c
kern_return_t thread_set_state
(mach_port_t target_thread,
 int flavor,
 thread_state_t new_state,
 mach_msg_type_number_t new_stateCnt);

kern_return_t thread_set_state_secure
(mach_port_t target_thread,
 int flavor,
 thread_state_t new_state,
 mach_msg_type_number_t new_stateCnt);
```

**DESCRIPTION**

The *thread_set_state* function sets the execution state (for example, the machine registers) for *target_thread*. *flavor* specifies the type of state to set. The *thread_set_state_secure* function changes the state of *target_thread*’s associated task structure from THREAD_Created to THREAD_STATE_SET.

The format of the state to set is machine specific; it is defined in `<mach/thread_status.h>`. For *thread_set_state_secure* the state may be limited to ensure that the new child task is started at a valid entry point.

**SECURITY**

For *thread_set_state* the requesting task must hold *thsv_set_thread_state* permission to *target_thread*. For *thread_set_state_secure* the requesting task must hold *thsv_set_thread_state_secure* and *tsv_initiate_secure* permission to *target_thread* and *target_thread*’s associated thread structure must be in the THREAD_CREATED state.

**PARAMETERS**

- **target_thread**
  [in thread port] The thread for which to set the execution state. The calling thread cannot specify itself.

- **flavor**
  [in scalar] The type of state to set. Valid values correspond to supported machine architecture features.
thread_set_state/thread_set_state_secure

new_state

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

new_stateCnt

[in scalar] The size of the state array (in units of sizeof (int)). The maximum size is defined by THREAD_STATE_MAX.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

**Function** — Suspends a thread

**SYNOPSIS**

```c
kern_return_t thread_suspend(mach_port_t target_thread);
```

**DESCRIPTION**

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

In this context, a user-level instruction can be either a machine instruction executed in user mode or a system trap instruction, including a page fault. If a thread is currently executing within a system trap, the kernel code may continue to execute until it reaches the system return code or it may suspend within the kernel code. In either case, the system trap returns when the thread resumes.

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

**SECURITY**

The requesting task must hold `thsv_suspend_thread` permission to `target_thread`.

**PARAMETERS**

- `target_thread` [in thread port] The thread to be suspended.

**CAUTIONS**

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

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

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

System Trap — Cause context switch with options

LIBRARY
Not declared anywhere.

SYNOPSIS

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

DESCRIPTION

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

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

SWITCH_OPTION_NONE
The time argument is ignored.

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

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

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

The requesting task must hold thsv_switch_thread and thsv_depress_pri permission to new_thread.

PARAMETERS

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

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

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

NOTES

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

CAUTIONS

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: swtch, swtch_pri, thread_abort, thread_depress_abort.
thread_terminate

Function — Destroys a thread

SYNOPSIS

kern_return_t thread_terminate
    (mach_port_t target_thread);

DESCRIPTION

The thread_terminate function kills creates target_thread.

SECURITY

The requesting task must hold thsv_terminate_thread permission to target_thread.

PARAMETERS

    target_thread
        [in thread port] The thread to be destroyed.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

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

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t thread_wire
    (mach_port_t host_priv, 
     mach_port_t thread, 
     boolean_t wired);

DESCRIPTION

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

SECURITY

The requesting task must hold hpsv_wire_thread permission to host_priv and thsv_wire_thread_into_memory to thread.

PARAMETERS

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

thread  
[in thread port] The thread to be wired.

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

RETURN VALUE

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

RELATED INFORMATION

Functions: \texttt{vm\_wire}.
Thread Interface
This chapter discusses the specifics of the kernel’s task interfaces. This includes functions that return status information for a task. Also included are functions that operate upon all or a set of threads within a task.
mach_ports_lookup

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

**SYNOPSIS**

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

**DESCRIPTION**

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

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

**SECURITY**

The requesting task must hold `tsv_lookup_ports` permission to `target_task`.

**PARAMETERS**

- `target_task`
  [in task port] The task whose currently registered ports are to be returned.

- `init_port_set`
  [out pointer to dynamic array of registered ports] The returned array of ports.

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

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `mach_ports_register`. 
mach_ports_register

**Function** — Registers an array of well-known system ports

**SYNOPSIS**

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

**DESCRIPTION**

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

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

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

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

**SECURITY**

The requesting task must hold `tsv_register_ports` permission to `target_task`.

**PARAMETERS**

- **target_task**
  
  [in task port] The task for which the ports are to be registered.

- **init_port_set**
  
  [in pointer to array of registered ports] The array of ports to register.

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

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

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

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: `mach_msg`, `mach_ports_lookup`. 
mach_task_self

System Trap — Returns the task self port

LIBRARY

#include <mach/mach_traps.h>

SYNOPSIS

mach_port_t mach_task_self
()

DESCRIPTION

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

SECURITY

The requesting task must hold tsv_get_task_kernel_port permission to the requesting task’s task port.

PARAMETERS

None

NOTES

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

RETURN VALUE

[task-self port] Send rights to the task’s port.

RELATED INFORMATION

Functions: task_info.
task_change_sid

**Function** — Changes the SID of a task

**SYNOPSIS**

```c
kern_return_t task_change_sid
    (mach_port_t target_task,
     security_id_t new_sid);
```

**DESCRIPTION**

The `task_change_sid` function changes the security identifier (SID) of `target_task` to `new_sid`. Currently, only the authentication identifier (AID) portion of the SID is allowed to change. Hence the mandatory identifier (MID) field of `new_sid` must be either 0 or the same as the MID field of the `target_task`'s SID.

**SECURITY**

The following permissions are required:

- the requesting task must hold `tsv_change_sid` permission to `target_task`'s task port
- the requesting task must hold `tsv_make_sid` permission to `new_sid`.
- The `target_task` must hold `tsv_transition_sid` to `new_sid`.

**PARAMETERS**

- `target_task`
  
  [in task port] The port for the task whose SID is being changed.

- `new_sid`
  
  [in security id] The new SID with which `target_task` will be labeled.

**RETURN VALUE**

Generic errors apply.

**RELATED INFORMATION**

Functions: None
task_create/task_create_secure

Function — Creates a task

SYNOPSIS

kern_return_t task_create
(mach_port_t parent_task,
  boolean_t inherit_memory,
  mach_port_t* child_task);

kern_return_t task_create_secure
(mach_port_t parent_task,
  boolean_t inherit_memory,
  mach_port_t* child_task,
  security_id_t subj_sid);

DESCRIPTION

The task_create and task_create_secure functions create a new task from parent_task and return the name of the new task in child_task. The child task acquires shared or copied parts of the parent’s address space (see vm_inherit). The child task initially contains no threads.

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

- **task_kernel_port** — The port by which the kernel knows the new child task. The child task holds a send right for this port. The port name is also returned to the calling task.

- **task_bootstrap_port** — The port to which the child task can send a message requesting return of any system service ports that it needs (for example, a port to the Network Name Server or the Environment Manager). The child task inherits a send right for this port from the parent task. The child task can use task_set_special_port to change this port.

- **task_exception_port** — A default exception port for the child task, inherited from the parent task. The exception port is the port to which the kernel sends exception messages. Exceptions are synchronous interruptions to the normal flow of program control caused by the program itself. Some exceptions are handled transparently by the kernel, but others must be reported to the program. The child task, or any one of its threads, can change the default exception port to take an active role in exception handling (see task_set_special_port or thread_set_special_port).

The child task also inherits the following ports:

- [sample port] The port to which PC sampling messages are to be sent.
- [registered ports] Ports to system services.
In addition to creating a new task, `task_create_secure` assigns the specified security identifier to the new task. Because of the necessity to control what the parent task may do to the child task via `child_task` the newly created task structure state is set to EMPTY to ensure that `thread_create`, `thread_set_state` and `thread_resume` sequence uses the secure variants of these requests. This assures the proper start up sequence upon a cross context task creation. `task_create` sets the created task structure state to TASK_READY and does not require special permissions to or processing sequences for the parent task to initiate processing in the child task.

**SECURITY**

For `task_create` the requesting task must hold `tsv_create_task` permission to `parent_task`. For `task_create_secure` the following permissions are required:

- the requesting task must hold `tsv_create_task_secure` permission to `parent_task`'s task port and  
- the requesting task must hold `tsv_cross_context_create` to `child_task`'s task port.  
- The `parent_task` must hold `tsv_cross_context_inherit` to `child_task`'s task port.

The permission to inherited memory in tasks created with the use of `task_create_secure` is as determined by the system’s security policy. It will be based on the relationship between the new task’s security identity and the security identifier associated with the memory.

For `task_create` and for the case where no subject security identifier is provided on a `task_create_secure`, the child task is created with a subject security identifier that is the same as `parent_task`'s subject security identifier.

**PARAMETERS**

`parent_task`  
[in task port] The port for the task from which to draw the child task’s port rights, resource limits, and address space.

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

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

`subj_sid`  
[in security id] The security identifier to be associated with the child task.
RETURN VALUE

Generic errors apply.

RELATED INFORMATION

Functions: task_get_special_port, task_resume, task_set_special_port, task_suspend, task_terminate, task_threads, thread_create, thread_create_secure, thread_resume, thread_resume_secure, thread_set_state, thread_set_state_secure, vm_inherit, task_sample, norma_task_create.
**task_get_emulation_vector**

**Function** — Return user-level handlers for system calls.

**SYNOPSIS**

```c
kern_return_t task_get_emulation_vector(
    mach_port_t task,
    int* vector_start,
    emulation_vector_t* emulation_vector,
    mach_msg_type_number_t* emulation_vector_count);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `tsv_get_emulation` permission to `task`.

**PARAMETERS**

- `task` [in task port] The port for the task for which the system call handler addresses are desired.
- `vector_start` [out scalar] The syscall number corresponding to the first element of `emulation_vector`.
- `emulation_vector` [out pointer to dynamic array of `vm_offset_t`] Pointer to the returned array of routine entrypoints for the system calls starting with syscall number `vector_start`.
- `emulation_vector_count` [out scalar] The number of entries filled by the kernel.

**NOTES**

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

**RETURN VALUE**

Only generic errors apply.
RELATED INFORMATION

Functions: task_set_emulation, task_set_emulation_vector.
**task_get_special_port**

**Function** — Returns a send right to a special port

**SYNOPSIS**

```c
kern_return_t task_get_special_port(mach_port_t task, int which_port, mach_port_t* special_port);
```

**task_get_bootstrap_port**

Macro form

```c
kern_return_t task_get_bootstrap_port(mach_port_t task, mach_port_t* special_port)
⇒ task_get_special_port(task, TASK_BOOTSTRAP_PORT, special_port)
```

**task_get_exception_port**

Macro form

```c
kern_return_t task_get_exception_port(mach_port_t task, mach_port_t* special_port)
⇒ task_get_special_port(task, TASK_EXCEPTION_PORT, special_port)
```

**task_get_kernel_port**

Macro form

```c
kern_return_t task_get_kernel_port(mach_port_t task, mach_port_t* special_port)
⇒ task_get_special_port(task, TASK_KERNEL_PORT, special_port)
```

**DESCRIPTION**

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

If one task has a send right for the kernel port of another task, it can use the port to perform kernel operations for the other task. Send rights for a kernel port normally are held only by the task to which the port belongs, or by the task’s parent task. Using the `mach_msg` function, however, any task can pass a send right for its kernel port to another task.
SECURITY
The requesting task must hold \texttt{tsv\_get\_task\_boot\_port}, \texttt{tsv\_get\_task\_exception\_port} or \texttt{tsv\_get\_task\_kernel\_port} permission to \textit{task} to get, respectively, the target task’s boot port, exception port or kernel port.

PARAMETERS
\begin{itemize}
  \item \texttt{task}  
    [in task port] The port for the task for which to return the port’s send right.
  \item \texttt{which\_port}  
    [in scalar] The special port for which the send right is requested. Valid values are:
    \begin{itemize}
      \item \texttt{TASK\_KERNEL\_PORT}  
        [task-self port] The port used to control this task. Used to send messages that affect the task. This is the port returned by \texttt{mach\_task\_self}.
      \item \texttt{TASK\_BOOTSTRAP\_PORT}  
        [bootstrap port] The task’s bootstrap port. Used to send messages requesting return of other system service ports.
      \item \texttt{TASK\_EXCEPTION\_PORT}  
        [exception port] The task’s exception port. Used to receive exception messages from the kernel.
    \end{itemize}
  \item \texttt{special\_port}  
    [out task-special port] The returned value for the port.
\end{itemize}

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: \texttt{mach\_task\_self}, \texttt{task\_create}, \texttt{task\_set\_special\_port}, \texttt{thread\_get\_special\_port}, \texttt{thread\_set\_special\_port}, \texttt{mach\_task\_self}.
task_info

Function — Returns information about a task

SYNOPSIS

kern_return_t task_info
    (mach_port_t target_task,
     int flavor,
     task_info_t task_info,
     mach_msg_type_number_t* task_infoCnt);

DESCRIPTION

The task_info function returns an information array of type flavor.

SECURITY

The requesting task must hold tsv_get_task_info permission to target_task.

PARAMETERS

    target_task
    [in task port] The port for the task for which the information is to be re-
    turned.

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

    TASK_BASIC_INFO
    Returns basic information about the task, such as the task’s
    suspend count and number of resident pages. The structure re-
    turned is task_basic_info, whose size is given by
    TASK_BASIC_INFO_COUNT.

    TASK_SECURE_INFO
    Returns basic information about the task, such as the task’s
    suspend count, number of resident pages and security identifi-
    er. The structure returned is task_basic_secure_info, whose
    size is given by TASK_BASIC_SECURE_INFO_COUNT.

    TASK_THREAD_TIMES_INFO
    Returns system and user space run-times for live threads. The
    structure returned is task_thread_times_info, whose size is
    given by TASK_THREAD_TIMES_INFO_COUNT.

    task_info
    [out array of int] Information about the specified task.
task_info

    task_infoCnt
    [pointer to in/out scalar] On input, the maximum size of the info buffer; on output, the returned size of the information structure (in units of sizeof (int)). The maximum size is defined by TASK_INFO_MAX.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Data Structures: task_basic_info, task_thread_times_info.
task_resume

Function — Resume a task

SYNOPSIS

kern_return_t task_resume
   (mach_port_t task);

DESCRIPTION

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

SECURITY

The requesting task must hold tsv_resume_task permission to task.

PARAMETERS

task
   [in task port] The port for the task to be resumed.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

**Function** — Perform periodic PC sampling for a task

**SYNOPSIS**

```c
kern_return_t task_enable_pc_sampling
  (mach_port_t task, int *ticks; sampled_pc_flavor_t flavor);
kern_return_t task_disable_pc_sampling
  (mach_port_t task, int *sample_cnt; sampled_pc_flavor_t flavor);
kern_return_t task_get_sampled_pcs
  (mach_port_t task, unsigned *seqno; sampled_pc_t sampled_pcs[], int *sample_cnt);
```

**DESCRIPTION**

These functions cause the program counter (PC) of the specified `task` to be sampled periodically (whenever one of the task’s threads happens to be running at the time of the kernel’s “hardclock” interrupt). The set of PC sample values obtained are saved in buffers.

**SECURITY**

These functions require that the requesting task hold `tsv_sample_task` permission to `task`.

**PARAMETERS**

- **thread**
  [in thread port] Thread whose PC is to be sampled

- **ticks**
  [out scalar] the kernel’s idea of clock granularity (ticks per second). Don’t trust this.

- **flavor**
  [in structure] The sampling flavor, which can be any of the following flavors defined in `pc_sample.h`.

  SAMPLED_PC_PERIODIC,
  SAMPLED_PC_VM_ZFILL_FAULTS,
SAMPL ED PC_VM.REACTIVATION_FAULTS,
SAMPL ED_PC_VM.PAGIN_FAULTS,
SAMPL ED_PC_VM.COM_FAULTS,
SAMPL ED_PC_VM.FAULTS.ANY,
SAMPL ED_PC_VM_FAULTS.

seqno
[out scalar] The sequence number of the sampled PC’s. This is useful for determining when a collector thread has missed a sample.

sampled_pcs
[out structure] The sampled PCs for threads in task. A sample contains three fields: a thread-specific unique identifier, a PC value and the type of sample as per flavor.

sample_cnt
[out scaler] The number of sample elements in the kernel for the named task or thread.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: thread_enable_pc_sampling, thread_disable_pc_sampling, thread_get_sampled_pcs.
**task_set_emulation**

**Function** — Establish a user-level handler for a system call.

**SYNOPSIS**

```c
kern_return_t task_set_emulation(
    mach_port_t task,
    vm_address_t routine_entry_pt,
    int syscall_number);
```

**DESCRIPTION**

The `task_set_emulation` function establishes a handler within the task for a particular system call. When a thread executes a system call with this particular number, the system call will be redirected to the specified routine within the task’s address space. This is expected to be an address within the transparent emulation library.

These emulation handler addresses are inherited by child processes.

**SECURITY**

The requesting task must hold `tsv_set_emulation` permission to `task`.

**PARAMETERS**

- `task` [in task port] The port for the task for which to establish the system call handler.
- `routine_entry_pt` [in scalar] The address within the task of the handler for this particular system call.
- `syscall_number` [in scalar] The number of the system call to be handled by this handler.

**NOTES**

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

**RETURN VALUE**

Only generic errors apply.
RELATED INFORMATION

Functions: task_set_emulation_vector, task_get_emulation_vector.
task_set_emulation_vector

**Function** — Establishes user-level handlers for system calls.

**SYNOPSIS**

```c
kern_return_t task_set_emulation_vector(
    mach_port_t task,
    int vector_start,
    emulation_vector_t emulation_vector,
    mach_msg_type_number_t emulation_vector_count);
```

**DESCRIPTION**

The `task_set_emulation_vector` function establishes a handler within the task for a set of system calls. When a thread executes a system call with one of these numbers, the system call will be redirected to the corresponding routine within the task’s address space. This is expected to be an address within the transparent emulation library.

These emulation handler addresses are inherited by child processes.

**SECURITY**

The requesting task must hold `tsv_set_emulation` permission to `task`.

**PARAMETERS**

- `task` [in task port] The port for the task for which to establish the system call handler.

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

- `emulation_vector` [in pointer to array of `vm_offset_t`] An array of routine entrypoints for the system calls starting with syscall number `vector_start`.


**NOTES**

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

Only generic errors apply.

RELATED INFORMATION

Functions: task_set_emulation, task_get_emulation_vector.
task_set_special_port

Function — Sets a special port for a task

SYNOPSIS

kern_return_t task_set_special_port
    (mach_port_t task,
     int which_port,
     mach_port_t special_port);

task_set_bootstrap_port

Macro form

kern_return_t task_set_bootstrap_port
    (mach_port_t task,
     mach_port_t special_port)
⇒ task_set_special_port (task, TASK_BOOTSTRAP_PORT, special_port)

task_set_exception_port

Macro form

kern_return_t task_set_exception_port
    (mach_port_t task,
     mach_port_t special_port)
⇒ task_set_special_port (task, TASK_EXCEPTION_PORT, special_port).

task_set_kernel_port

Macro form

kern_return_t task_set_kernel_port
    (mach_port_t task,
     mach_port_t special_port)
⇒ task_set_special_port (task, TASK_KERNEL_PORT, special_port)

DESCRIPTION

The task_set_special_port function sets a special port belonging to task.

SECURITY

The requesting task must hold tsv_set_task_boot_port, tsv_set_task_exception_port or tsv_set_task_kernel_port permission to task to set, respectively, task’s boot port, exception port or kernel port.

PARAMETERS

task
    [in task port] The task for which to set the port.
**Task Interface**

*which_port*
- [in scalar] The special port to be set. Valid values are:
  - **TASK_BOOTSTRAP_PORT**
    - [bootstrap port] The task’s bootstrap port. Used to send messages requesting return of other system service ports.
  - **TASK_EXCEPTION_PORT**
    - [exception port] The task’s exception port. Used to receive exception messages from the kernel.
  - **TASK_KERNEL_PORT**
    - [task-self port] The task’s kernel port. Used by the kernel to receive messages from the task. This is the port returned by `mach_task_self`.

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

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `task_create`, `task_get_special_port`, `exception_raise`, `mach_task_self`, `thread_get_special_port`, `thread_set_special_port`. 
task_suspend

Function — Suspending a task

SYNOPSIS

kern_return_t task_suspend
    (mach_port_t task);

DESCRIPTION

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

SECURITY

The requesting task must hold tsv_suspend_task permission to task.

PARAMETERS

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

*Function* — Destroys a task

**SYNOPSIS**

```c
kern_return_t task_terminate
    (mach_port_t task);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold *tsv_terminate_task* permission to *task*.

**PARAMETERS**

*task*  
[In task port] The port for the task to be destroyed.

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

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

Function — Returns a list of the threads within a task

SYNOPSIS

kern_return_t task_threads
      (mach_port_t task,
       thread_array_t* thread_list,
       mach_msg_type_number_t* thread_count);

DESCRIPTION

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

SECURITY

The requesting task must hold tsv_get_task_threads permission to task.

PARAMETERS

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: thread_create, thread_terminate, thread_suspend.
Task Interface
This chapter discusses the specifics of the kernel’s host interfaces. Included are functions that return status information for a host, such as kernel statistics.

Note that hosts are named both by a name port, which allows the holder to request information about the host, and a control port, which provides full control access. The control port for a host is provided to the bootstrap task for that host.
host_adjust_time

**Function** — Gradually change the time

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

```c
kern_return_t host_adjust_time
    (mach_port_t host_priv,
     time_value_t new_adjustment,
     time_value_t* old_adjustment);
```

**DESCRIPTION**

The `host_adjust_time` function arranges for the time on a specified host to be gradually changed by an adjustment value.

**SECURITY**

The requesting task must hold `hpsv_set_time` permission to `host_priv`.

**PARAMETERS**

- `host_priv`  
  [in host-control port] The control port the host for which the time is to be set.

- `new_adjustment`  
  [in structure] New adjustment value.

- `old_adjustment`  
  [out structure] Old adjustment value.

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `host_get_time`, `host_set_time`.

Data Structures: `time_value`. 
host_get_boot_info

**Function** — Return operator boot information

**LIBRARY**

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

**SYNOPSIS**

```
kern_return_t host_get_boot_info
    (mach_port_t priv_host,
     kernel_boot_info_t boot_info);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `hpsv_get_boot_info` permission to `priv_host`.

**PARAMETERS**

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

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

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `host_info`.
host_get_special_port

**Function** — Return a send right to a special port

**LIBRARY**

```
#include <mach.h>
```

**SYNOPSIS**

```
kern_return_t host_get_special_port
    (mach_port_t host,
     int port_label,
     mach_port_t* special_port);
```

**DESCRIPTION**

The `host_get_special_port` function returns a send right to `special_port` as requested in `port_label`.

**SECURITY**

The requesting task must have `hsv_get_special_port` to `host`. Depending on the value of `port_label`, the requesting task must also hold one of the following permissions to `host`:

- `hsv_get_audit_port`
- `hsv_get_authentication_port`
- `hsv_get_crypto_port`
- `hsv_get_host_control_port`
- `hsv_get_negotiation_server_port`
- `hsv_get_network_server_port`
- `hsv_get_security_master_port`
- `hsv_get_security_client_port`

`hsv_get_host_control_port` is also used to control access to the master device port.

**PARAMETERS**

- `host_name_port`
  [in host-name port] The host name port to which the request is sent.

- `port_label`
  [in scalar] Specifies which special port the function should return. This parameter can take on one of the following values:
host_get_special_port

- AUDIT_SERVER_PORT
- AUTHENTICATION_SERVER_PORT
- CRYPTO_SERVER_PORT
- HOST_CONTROL_PORT
- MASTER_DEVICE_PORT
- NEGOTIATION_SERVER_PORT
- NETWORK_SECURITY_SERVER_PORT
- SECURITY_SERVER_CLIENT_PORT
- SECURITY_SERVER_MASTER_PORT

special_port
  [out port] A send right to the requested port.

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: host_set_special_port.
host_get_time

Function --- Return the current time.

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t host_get_time
     (mach_port_t host,
      time_value_t* current_time);

DESCRIPTION

The host_get_time function returns the current time as seen by that host.

SECURITY

The requesting task must hold hsv_get_time permission to host.

PARAMETERS

host
     [in host-name port] The name port of the host from which the time is
     to be obtained.

current_time
     [out structure] Returned time value.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: host_adjust_time, host_set_time.

Data Structures: time_value.
**host_info**

**Function** — Returns information about a host

**LIBRARY**

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

**SYNOPSIS**

```
kern_return_t host_info
    (mach_port_t host,
     int flavor,
     host_info_t host_info,
     mach_msg_type_number_t* host_infoCnt);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `hsv_get_host_info` permission to `host`.

**PARAMETERS**

host

[in host-name port] The name port for the host for which information is to be obtained.

flavor

[in scalar] The type of statistics desired.

**HOST_BASIC_INFO**

Basic information (number of processors, amount of memory). The returned structure is `host_basic_info` of size `HOST_BASIC_INFO_COUNT`.

**HOST_LOAD_INFO**

Scheduling statistics. The returned structure is `host_load_info` of size `HOST_LOAD_INFO_COUNT`.

**HOST_PROCESSOR_SLOTS**

An array of the processor slot numbers (natural-sized units) for active processors.
Host Interface

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

host_info
[out array of int] Statistics about the specified host.

host_infoCnt
[pointer to in/out scalar] On input, the maximum size of the info buffer; on output, the size of the information structure (in units of sizeof (int)).

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

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: host_get_boot_info, host_kernel_version, host_processors, processor_info.

Data Structures: host_basic_info, host_load_info, host_sched_info
host_kernel_version

**Function** — Returns kernel version information for a host

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

```c
kern_return_t host_kernel_version(mach_port_t host,
                                 kernel_version_t version);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `hsv_get_host_version` permission to `host`.

**PARAMETERS**

- `host` [in host-name port] The name port for the host for which information is to be obtained.
- `version` [out array of char] Character string describing the kernel version executing on `host`.

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `host_info`. 
**host_reboot**

**Function** — Reboot this host

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

```c
kern_return_t host_reboot(mach_port_t host_priv, int options);
```

**DESCRIPTION**

The `host_reboot` function reboots the specified host.

**SECURITY**

The requesting task must hold `hpsv_reboot_host` permission to `host_priv`.

**PARAMETERS**

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

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

**NOTES**

If successful, this call will not return.

**RETURN VALUE**

Only generic errors apply.
host_set_special_port

Function — Sets special kernel ports

LIBRARY

#include <mach.h>

SYNOPSIS

kern_return_t host_set_special_port
    (mach_port_t host,
     int port_label,
     mach_port_t port_value);

DESCRIPTION

The host_set_special_port function supplies a port to the specified host for use as the port selected by the port_label.

SECURITY

The requesting task must have hsv_set_special_port to host. Depending on the value of port_label, the requesting task must also hold one of the following permissions to host:

- hsv_set_audit_port
- hsv_set_authentication_port
- hsv_set_crypto_port
- hsv_set_negotiation_port
- hsv_set_network_ss_port
- hsv_set_security_master_port
- hsv_set_security_client_port

PARAMETERS

host
    [in host-name port] The name port for the host for which the specified port will be set.

port_label
    [in scalar] A label for which the special port will be set. This parameter can take on one of the following values:

    • AUDIT_SERVER_PORT
    • AUTHENTICATION_SERVER_PORT
    • CRYPTO_SERVER_PORT
Host Interface

- NEGOTIATION_SERVER_PORT
- NETWORK_SECURITY_SERVER_PORT
- SECURITY_SERVER_MASTER_PORT
- SECURITY_SERVER_CLIENT_PORT

port_value
[in port] A port for the kernel to use for the selected operation.

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: host_get_special_port
host_set_time

  Function — Sets the time

LIBRARY
  
  #include <mach/mach_host.h>

SYNOPSIS
  
  kern_return_t host_set_time
      (mach_port_t host_priv,
       time_value_t new_time);

DESCRIPTION
  
  The host_set_time function establishes the time on the specified host.

SECURITY
  
  The requesting task must hold hpsv_set_time permission to host_priv.

PARAMETERS
  
  host_priv
    [in host-control port] The control port for the host for which the time is
to be set.

  new_time
    [in structure] Time to be set.

RETURN VALUE
  
  Only generic errors apply.

RELATED INFORMATION
  
  Functions: host_adjust_time, host_get_time.

  Data Structures: time_value.
mach_host_self

System Trap — Returns the host self port

LIBRARY
#include <mach/mach_traps.h>

SYNOPSIS
mach_port_t mach_host_self
();

SECURITY
The requesting task must hold hsv_get_host_name permission to the processor's host name port.

DESCRIPTION
The mach_host_self function returns send rights to the current host’s name port.

PARAMETERS
None

RETURN VALUE
[host-name port] Send rights to the host’s name port.

RELATED INFORMATION
Functions: host_info.
This chapter discusses the specifics of the kernel’s processor and processor set interfaces. This includes functions to control processors, change their assignments, assign tasks and threads to processors, and processor status returning functions.

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

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

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

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

```c
kern_return_t host_processor_set_priv(
    mach_port_t host_priv,
    mach_port_t set_name,
    mach_port_t* processor_set);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `hpsv_pset_ctrl_port` permission to `host_priv`.

**PARAMETERS**

- `host_priv` [in host-control port] The control port for the host for which the processor set is desired.
- `set_name` [in processor-set-name port] The name port for the processor set desired.

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

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

**Function** — Returns processor set ports for a host

**LIBRARY**

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

**SYNOPSIS**

```
kern_return_t host_processor_sets
    (mach_port_t host,
     processor_set_name_array_t* processor_set_list,
     mach_msg_type_number_t* processor_set_count);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `hsv_pset_names` permission to `host`.

**PARAMETERS**

- **host**
  [in host-name port] The name port for the host for which the processor sets are desired.

- **processor_set_list**
  [out pointer to dynamic array of processor-set-name ports] The set of processor set name ports for those currently existing on `host`; no particular order is guaranteed.

- **processor_set_count**
  [out scalar] The number of processor set names returned.

**NOTES**

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

`processor_set_list` is automatically allocated by the kernel, as if by `vm_allocate`. It is good practice to `vm_deallocate` this space when it is no longer needed.
RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: `host_processor_set_priv`, `processor_set_create`, `processor_set_tasks`, `processor_set_threads`. 
host_processors

Function — Gets processor ports for a host

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t host_processors
    (mach_port_t host_priv,
     processor_array_t* processor_list,
     mach_msg_type_number_t* processor_count);

DESCRIPTION

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

SECURITY

The requesting task must hold hpsv_get_host_processors permission to host_priv.

PARAMETERS

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: processor_start, processor_exit, processor_info, processor_control.
### processor_assign

**Function** — Assign a processor to a processor set

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

```c
kern_return_t processor_assign(mach_port_t processor, mach_port_t new_set, boolean_t wait);
```

**DESCRIPTION**

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

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

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

**SECURITY**

The requesting task must hold `psv_assign_processor_to_set` permission to `processor` and `pssv_assign_processor` to `new_set`.

**PARAMETERS**

- **processor**
  - [in processor port] The processor to be assigned.

- **new_set**
  - [in processor-set-control port] The control port for the processor set into which the processor is to be assigned.
**processor_assign**

*wait*

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

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `processor_set_create`, `processor_set_info`, `task_assign`, `thread_assign`.
processor_control

Function — Do something to a processor

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t processor_control (mach_port_t processor, processor_info_t cmd, mach_msg_type_number_t count);

DESCRIPTION

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

SECURITY

The requesting task must hold psv_may_control_processor permission to processor.

PARAMETERS

processor

[in processor port] The processor to be controlled.

cmd

[pointer to in array of int] An array containing the command to be applied to the processor.

count

[in scalar] The size of the cmd array.

NOTES

These operations are machine dependent. They may do nothing.

RETURN VALUE

KERN_FAILURE

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

Functions: processor_start, processor_exit, processor_info, host_processors.
processor_exit

Function — Exit a processor

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t processor_exit
    (mach_port_t processor);

DESCRIPTION

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

SECURITY

The requesting task must hold psv_may_control_processor permission to processor.

PARAMETERS

processor
    [in processor port] The processor to be controlled.

NOTES

This operation is machine dependent. It may do nothing.

CAUTIONS

The ability to restart an exited processor is machine dependent.

RETURN VALUE

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

RELATED INFORMATION

Functions: processor_control, processor_start, processor_info, host_processors.
processor_get_assignment

Function — Get current assignment for a processor

LIBRARY
#include <mach/mach_host.h>

SYNOPSIS
kern_return_t processor_get_assignment
          (mach_port_t processor,
           mach_port_t* assigned_set);

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

SECURITY
The requesting task must hold psv_get_processor_assignment permission to processor.

PARAMETERS

processor
[in processor port] The processor whose assignment is desired.

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

RETURN VALUE
KERN_FAILURE
processor is either shut down or off-line.

RELATED INFORMATION
Functions: processor_assign, processor_set_create, processor_info, task_assign, thread_assign.
processor_info

**Function** — Returns information about a processor.

**LIBRARY**

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

**SYNOPSIS**

```
kern_return_t processor_info(
    mach_port_t processor,               // in processor port
    int flavor,                          // in scalar
    mach_port_t* host,                  // out host-name port
    processor_info_t processor_info,    // out array of int
    mach_msg_type_number_t* processor_infoCnt);
```

**DESCRIPTION**

The `processor_info` function returns selected information for a processor as an array, as specified by `flavor`.

**SECURITY**

The requesting task must hold `psv_get_processor_info` permission to `processor`.

**PARAMETERS**

- `processor`
  [in processor port] A processor port for which information is desired.

- `flavor`
  [in scalar] The type of information requested.

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

- `processor_info`
  [out array of int] Information about the processor.
processor_info

processor_infoCnt
[pointer to in/out scalar] On input, the maximum size of the info buffer; on output, the returned size of the info structure (in units of sizeof (int)).

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: processor_start, processor_exit, processor_control, host_processors.

Data Structures: processor_basic_info.
processor_set_create

**Function** — Creates a new processor set

**LIBRARY**

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

**SYNOPSIS**

```
kern_return_t processor_set_create
    (mach_port_t host_name,
     mach_port_t* new_set,
     mach_port_t* new_name);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `hsv_create_pset` permission to `host_name`.

**PARAMETERS**

- **host_name**
  [in host-name port] The name port for the host on which the set is to be created.

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

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

**RETURN VALUE**

Only generic errors apply.
processor_set_create

RELATED INFORMATION
Functions: processor_set_destroy, processor_set_info, processor_assign, task_assign, thread_assign.
processor_set_default

**Function** — Returns the default processor set

**LIBRARY**

`#include <mach/mach_host.h>`

**SYNOPSIS**

```c
kern_return_t processor_set_default(mach_port_t host, mach_port_t* default_set_name);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `psw_get_default_pset_name` permission to `host`.

**PARAMETERS**

- **host**
  
  [in host-name port] The name port for the host for which the default processor set is desired.

- **default_set_name**
  
  [out processor-set-name port] The returned name port for the default processor set.

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `processor_set_info`, `thread_assign`, `task_assign`. 
processor_set_destroy

Function — Destroys a processor set

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t processor_set_destroy (mach_port_t processor_set);

DESCRIPTION

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

SECURITY

The requesting task must hold pssv_destroy_pset permission to processor_set.

PARAMETERS

processor_set

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

RETURN VALUE

KERN_FAILURE

An attempt was made to destroy the default processor set.

RELATED INFORMATION

Functions: processor_set_create, processor_assign, task_assign, thread_assign.
processor_set_info

Function — Returns information about a processor set.

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t processor_set_info
  (mach_port_t processor_set_name,
   int flavor,
   mach_port_t* host,
   processor_set_info_t processor_set_info,
   mach_msg_type_number_t* infoCnt);

DESCRIPTION

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

SECURITY

The requesting task must hold pssv_get_pset_info permission to processor_set_name.

PARAMETERS

processor_set_name
  [in processor-set-control port] A processor set control port for which information is desired.

flavor
  [in scalar] The type of information requested.

PROCESSOR_SET_BASIC_INFO
  Basic information concerning the processor set. The returned structure is defined by processor_set_basic_info, whose size is defined by PROCESSOR_SET_BASIC_INFO_COUNT.

PROCESSOR_SET_SCHED_INFO
  Scheduling information. The returned structure is defined by processor_set_sched_info, whose size is defined by PROCESSOR_SET_SCHED_INFO_COUNT.
**host**

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

**processor_set_info**

[out array of int] Information about the processor set.

**infoCnt**

[pointer to in/out scalar] On input, the maximum size of the info buffer; on output, the returned size of the info structure (in units of sizeof (int)).

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `processor_set_create`, `processor_set_default`, `processor_assign`, `task_assign`, `thread_assign`.

Data Structures: `processor_set_basic_info`, `processor_set_sched_info`. 
**processor_set_max_priority**

**Function** — Sets the maximum scheduling priority for a processor set

**LIBRARY**

`#include <mach/mach_host.h>`

**SYNOPSIS**

```c
kern_return_t processor_set_max_priority(
    mach_port_t processor_set,
    int priority,
    boolean_t change_threads);
```

**DESCRIPTION**

The `processor_set_max_priority` function sets the maximum scheduling priority for `processor_set`. The maximum priority of a processor set is used only when creating new threads. A new thread’s maximum priority is set to that of its assigned processor set. When assigned to a processor set, a thread’s maximum priority is reduced, if necessary, to that of its new processor set; its current priority is also reduced, as needed. Changing the maximum priority of a processor set does not affect the priority of the currently assigned threads unless `change_threads` is TRUE. If this priority change violates the maximum priority of some threads, their maximum priorities will be reduced to match.

**SECURITY**

The requesting task must hold `pssv_chg_pset_max_pri` permission to `processor_set`.

**PARAMETERS**

- `processor_set`  
  [in processor-set-control port] The control port for the processor set whose maximum scheduling priority is to be set.

- `priority`  

- `change_threads`  
  [in scalar] True if the maximum priority of existing threads assigned to this processor set should also be changed.

**RETURN VALUE**

Only generic errors apply.
processor_set_max_priority

RELATED INFORMATION

Functions: thread_max_priority, thread_priority, thread_assign.
**processor_set_policy_disable**

Function — Disables a scheduling policy for a processor set

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

```c
kern_return_t processor_set_policy_disable
    (mach_port_t processor_set,
     int policy,
     boolean_t change_threads);
```

**DESCRIPTION**

The `processor_set_policy_disable` function restricts the set of scheduling policies allowed for `processor_set`. The set of scheduling policies allowed for a processor set is the set of policies allowed to be set for threads assigned to that processor set. The current set of permitted policies can be obtained from `processor_set_info`. Timesharing may not be forbidden for any processor set. This is a compromise to reduce the complexity of the assign operation; any thread whose policy is forbidden by its target processor set has its policy reset to timesharing. Disabling a scheduling policy for a processor set has no effect on threads currently assigned to that processor set unless `change_threads` is TRUE, in which case their policies will be reset to timesharing.

**SECURITY**

The requesting task must hold `pssv_invalidate_scheduling_policy` permission to `processor_set`.

**PARAMETERS**

- **processor_set**
  
  [in processor-set-control port] The control port for the processor set for which a scheduling policy is to be disabled.

- **policy**
  
  [in scalar] Policy to be disabled. The values currently defined are `POLICY_TIMESHARE` and `POLICY_FIXEDPRI`.

- **change_threads**
  
  [in scalar] If true, causes the scheduling policy for all threads currently running with `policy` to `POLICY_TIMESHARE`. 

processor_set_policy_disable

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: processor_set_policy_enable, thread_policy.
processor_set_policy_enable

Function — Enables a scheduling policy for a processor set

LIBRARY
#include <mach/mach_host.h>

SYNOPSIS

kern_return_t processor_set_policy_enable
    (mach_port_t processor_set,
     int policy);

DESCRIPTION
The processor_set_policy_enable function extends the set of scheduling policies allowed for processor_set. The set of scheduling policies allowed for a processor set is the set of policies allowed to be set for threads assigned to that processor set. The current set of permitted policies can be obtained from processor_set_info.

SECURITY
The requesting task must hold pssv_define_new_scheduling_policy permission to processor_set.

PARAMETERS

processor_set
[in processor-set-control port] The control port for the processor set for which a scheduling policy is to be enabled.

policy
[in scalar] Policy to be enabled. The values currently defined are POLICY_TIMESHARE and POLICY_FIXEDPRI.

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: processor_set_policy_disable, thread_policy.
processor_set_tasks

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

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t processor_set_tasks
    (mach_port_t processor_set,
     task_array_t* task_list,
     mach_msg_type_number_t* task_count);

DESCRIPTION

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

SECURITY

The requesting task must hold pssv_observe_pset_processes permission to processor_set.

PARAMETERS

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: processor_set_threads, task_assign, thread_assign.
processor_set_threads

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

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t processor_set_threads
mach_port_t processor_set,
thread_array_t* thread_list,
mach_msg_type_number_t* thread_count);

DESCRIPTION

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

SECURITY

The requesting task must hold pssv_observe_pset_processes permission to processor_set.

PARAMETERS

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: processor_set_tasks, task_assign, thread_assign.
processor_start

Function — Start a processor

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t processor_start (mach_port_t processor);

DESCRIPTION

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

SECURITY

The requesting task must hold psv_may_control_processor permission to processor.

PARAMETERS

processor

[in processor port] The processor to be controlled.

NOTES

This operation is machine dependent. It may do nothing.

CAUTIONS

The ability to restart an exited processor is machine dependent.

RETURN VALUE

KERN_FAILURE

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

RELATED INFORMATION

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

**Function** — Assign a task to a processor set

**LIBRARY**

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

**SYNOPSIS**

```
kern_return_t task_assign
    (mach_port_t task,
     mach_port_t processor_set,
     boolean_t assign_threads);
```

**DESCRIPTION**

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

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

**SECURITY**

The requesting task must hold `tsv_assign_task_to_pset` permission to `task` and `pssv_assign_task` to `processor_set`.

**PARAMETERS**

- `task`
  - [in task port] The port for the task to be assigned.
- `processor_set`
  - [in processor-set-control port] The control port for the processor set into which the task is to be assigned.
- `assign_threads`
  - [in scalar] True if this assignment should apply as well to the threads within the task.

**RETURN VALUE**

Only generic errors apply.
RELATED INFORMATION

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

**Function** — Assign a task to the default processor set

**LIBRARY**

```
#include <arch/mach/mach_host.h>
```

**SYNOPSIS**

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

**DESCRIPTION**

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

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

**SECURITY**

The requesting task must hold `tsv_assign_task_to_pset` permission to `task` and `pssv_assign_task` permission to the default processor set.

**PARAMETERS**

- `task`
  [in task port] The port for the task to be assigned.

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

**NOTES**

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

**RETURN VALUE**

Only generic errors apply.
RELATED INFORMATION

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

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

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

```c
kern_return_t task_get_assignment
    (mach_port_t task,
     mach_port_t* processor_set);
```

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `tsv_get_task_assignment` permission to `task`.

**PARAMETERS**

- `task`
  [in task port] The port for the task whose assignment is desired.

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

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `task_assign`, `task_assign_default`, `processor_set_create`, `processor_set_info`, `thread_assign`, `processor_assign`.
task_priority

Function — Sets the scheduling priority for a task

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t task_priority
    (mach_port_t task,
     int priority,
     boolean_t change_threads);

DESCRIPTION

The task_priority function sets the scheduling priority for task. The priority of a task is used only when creating new threads. A new thread’s priority is set to that of the enclosing task’s priority. Changing the priority of a task does not affect the priority of the enclosed threads unless change_threads is TRUE. If this priority change violates the maximum priority of some threads, as many threads as possible will be changed and an error code will be returned.

SECURITY

The requesting task must hold tsv_chg_task_priority permission to task.

PARAMETERS

    task
        [in task port] The task whose scheduling priority is to be set.

    priority
        [in scalar] The new priority for the task.

    change_threads
        [in scalar] True if priority of existing threads within the task should also be changed.

RETURN VALUE

    KERN_FAILURE
        change_threads was TRUE and the attempt to change the priority of some existing thread within the task failed because the new priority would violate that thread’s maximum priority.
RELATED INFORMATION

Functions: thread_max_priority, thread_priority,
           processor_set_max_priority.
thread_assign

Function — Assign a thread to a processor set

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t thread_assign
  (mach_port_t thread,
   mach_port_t processor_set);

DESCRIPTION

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

SECURITY

The requesting task must hold thsv_assign_thread_to_pset permission to thread and pssv_assign_thread to processor_set.

PARAMETERS

thread
  [in thread port] The thread to be assigned.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Function — Assign a thread to the default processor set

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

kern_return_t thread_assign_default
(mach_port_t thread);

DESCRIPTION

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

SECURITY

The requesting task must hold thsv_assign_thread_to_pset permission to thread and pssv_assign_thread permission to the default processor set.

PARAMETERS

thread
[in thread port] The thread to be assigned.

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

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

LIBRARY
   #include <mach/mach_host.h>

SYNOPSIS
   kern_return_t thread_get_assignment
       (mach_port_t thread,
        mach_port_t* processor_set);

DESCRIPTION
   The thread_get_assignment function returns the name port to the processor set
to which thread is currently assigned. This port can only be used to obtain infor-
mation about the processor set.

SECURITY
   The requesting task must hold thsv_get_thread_assignment permission to
   thread.

PARAMETERS
   thread
       [in thread port] The thread whose assignment is desired.

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

RETURN VALUE
   Only generic errors apply.

RELATED INFORMATION
   Functions: thread_assign, thread_assign_default, processor_set_create,
               processor_set_info, task_assign, processor_assign.
**thread_max_priority**

**Function** — Sets the maximum scheduling priority for a thread

**LIBRARY**

`#include <mach/mach_host.h>`

**SYNOPSIS**

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

**DESCRIPTION**

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

Threads have three priorities associated with them by the system:

- A priority value which can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task.
- A maximum priority value which can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set.
- A scheduled priority value which is used to make scheduling decisions for the thread. This value is determined on the basis of the user priority value by the scheduling policy (for timesharing, this means adding an increment derived from CPU usage).

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

**SECURITY**

The requesting task must hold `thsv_set_max_thread_priority` permission to `thread`.

**PARAMETERS**

- `thread` [in thread port] The thread whose maximum scheduling priority is to be set.
thread_max_priority

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

priority

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: thread_priority, thread_policy, task_priority, processor_set_max_priority.
thread_policy

Function — Sets the scheduling policy to apply to a thread

LIBRARY

#include <mach/mach_host.h>

SYNOPSIS

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

DESCRIPTION

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

SECURITY

The requesting task must hold thsv_set_thread_policy permission to thread.

PARAMETERS

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

    policy
    [in scalar] Policy to be set. The values currently defined are POLICY_TIMESHARE and POLICY_FIXEDPRI.

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

RETURN VALUE

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

RELATED INFORMATION

Functions: processor_set_policy_enable, processor_set_policy_disable.
**thread_priority**

**Function** — Sets the scheduling priority for a thread

**LIBRARY**

```c
#include <mach/mach_host.h>
```

**SYNOPSIS**

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

**DESCRIPTION**

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

**SECURITY**

The requesting task must hold `thsv_set_thread_priority` to `thread`. If `set_max` is true, the requesting task must also hold `thsv_set_max_thread_priority` to `thread`.

**PARAMETERS**

- **thread**
  - [in thread port] The thread whose scheduling priority is to be set.

- **priority**

- **set_max**
  - [in scalar] True if the thread’s maximum priority should also be set.

**NOTES**

Threads have three priorities associated with them by the system:

- A priority value which can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task.

- A maximum priority value which can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set.

- A scheduled priority value which is used to make scheduling decisions for the thread. This value is determined on the basis of the user priority value by
the scheduling policy (for timesharing, this means adding an increment derived from CPU usage).

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

RETURN VALUE

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

RELATED INFORMATION

Functions: thread_max_priority, thread_policy, task_priority, processor_set_max_priority.
This chapter discusses the specifics of the device interfaces to in-kernel device drivers. These interfaces provide read, write and status interfaces to devices.
device_close

**Function** — De-establish a connection to a device.

**LIBRARY**

```c
#include <device/device.h>
```

**SYNOPSIS**

```c
kern_return_t device_close
  (mach_port_t device);
```

**DESCRIPTION**

The `device_close` function decrements the open count for the named device. If this count reaches zero, the close operation of the device driver is invoked, closing the device.

**SECURITY**

The requesting task must hold `dsv_close_device` permission to `device`.

**PARAMETERS**

- `device`
  
  [in device port] A device port to the device to be closed.

**RETURN VALUE**

- `D_NO_SUCH_DEVICE`
  No device with that name, or the device is not operational.

**RELATED INFORMATION**

Functions: `device_open`. 
device_get_status

Function — Return the current device status

LIBRARY

#include <device/device.h>

SYNOPSIS

kern_return_t device_get_status
(mach_port_t device,
int flavor,
dev_status_t status,
mach_msg_type_number_t* status_count);

DESCRIPTION

The device_get_status function returns status information pertaining to an open device. The possible values for flavor as well as the meaning of the returned status information is device dependent.

SECURITY

The requesting task must hold dsv_get_device_status permission to device.

PARAMETERS

device
[in device port] A device port to the device to be interrogated.

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

status
[out array of int] The returned device status.

status_count
[pointer to in/out scalar] On input, the reserved size of status; on output, the size of the returned device status.

RETURN VALUE

D_DEVICE_DOWN
Device has been shut down
Kernel Device Interface

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

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

RELATED INFORMATION
Functions: device_set_status.
**device_map**

*Function* — Establish a memory manager representing a device

**LIBRARY**

```c
#include <device/device.h>
```

**SYNOPSIS**

```c
kern_return_t device_map(mach_port_t device, vm_prot_t prot, vm_offset_t offset, vm_size_t size, mach_port_t* pager, int unmmap);
```

**DESCRIPTION**

The `device_map` function establishes a memory manager that presents a memory object representing a device. The resulting port is suitable for use as the memory manager port in a `vm_map` call. This call is device dependent.

**SECURITY**

The requesting task must hold `dsv_map_device` permission to `device`.

**PARAMETERS**

- `device` [in device port] A device port to the device to be mapped.
- `prot` [in scalar] Protection for the device memory.
- `offset` [in scalar] An offset within the device memory object, in bytes.
- `size` [in scalar] The size of the device memory object.
- `pager` [out abstract-memory-object port] The returned abstract memory object port to a memory manager that represents the device.
NOTES

Port rights are maintained as follows:

Abstract memory object port:
The device pager has all rights.

Memory cache control port:
The device pager has only send rights.

Memory cache name port:
The device pager has only send rights. The name port is not even recorded.

Regardless of how the object is created, the control and name ports are created by the kernel and passed through the memory management interface.

CAUTIONS

The device memory manager assumes that access to its memory objects will not be propagated to more than one host, and therefore provides no consistency guarantees beyond those made by the kernel.

In the event that more than one host attempts to use a device memory object, the device pager will only record the last set of port names. [This can happen with only one host if a new mapping is being established while termination of all previous mappings is taking place.] Currently, the device pager assumes that its clients adhere to the initialization and termination protocols in the memory management interface; otherwise, port rights or out-of-line memory from erroneous messages may be allowed to accumulate.

RETURN VALUE

D_DEVICE_DOWN
Device has been shut down

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

D_READ_ONLY
Data cannot be written to this device.

RELATED INFORMATION

Functions: vm_map, evc_wait.
device_open

**Function** — Establish a connection to a device.

**LIBRARY**

#include <device/device.h> (device_open)

#include <device/device_request.h> (device_open_request)

#include <device/device_reply.h> (ds_device_open_reply)

**SYNOPSIS**

```c
kern_return_t device_open
(mach_port_t master_port,
dev_mode_t mode,
dev_name_t name,
mach_port_t* device);
```

**device_open_request**

Asynchronous Function form — Asynchronously request a connection to a device

```c
kern_return_t device_open_request
(mach_port_t master_port,
mach_port_t reply_port,
dev_mode_t mode,
dev_name_t name);
```

**ds_device_open_reply**

Asynchronous Server Interface form — Receive the reply from an asynchronous open

```c
kern_return_t ds_device_open_reply
(mach_port_t reply_port,
kern_return_t return_code,
mach_port_t device);
```

**DESCRIPTION**

The device_open function opens a device object. The open operation of the device is invoked, if the device is not already open. The open count for the device is incremented.

**SECURITY**

The requesting task must hold dsv_open_device permission to master_port.
PARAMETERS

master_port
[in device-master port] The master device port. This port is provided to
the bootstrap task.

reply_port
[in reply port] The port to which a reply is to be sent when the device
is open.

mode
[in scalar] Opening mode. This is the bit-wise OR of the following val-
ues:

D_READ      Read access
D_WRITE     Write access
D_NODELAY   Do not delay on open

name
[pointer to in array of char] Name of the device to open.

return_code

device
[out device port] The returned device port.

RETURN VALUE

device_open_request returns only message transmission errors. The return val-
ue supplied to ds_device_open_reply is irrelevant. The return_code returned
by ds_device_open_reply or the error return from device_open is one of the
following:

D_WOULD_BLOCK
The device is busy, but D_NOWAIT was specified in mode.

D_ALREADY_OPEN
The device is already open in a mode incompatible with mode.

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.
D_DEVICE_DOWN
The device has been shut down.

D_READ_ONLY
Data cannot be written to this device.

RELATED INFORMATION
Functions: device_close, device_reply_server.
device_read

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

**LIBRARY**

```c
#include <device/device.h> (device_read)
```

```c
#include <device/device_request.h> (device_read_request)
```

```c
#include <device/device_reply.h> (ds_device_read_reply)
```

**SYNOPSIS**

```c
kern_return_t device_read
    (mach_port_t device,
     dev_mode_t mode,
     recnum_t recnum,
     int bytes_wanted,
     io_buf_ptr_t* data,
     mach_msg_type_number_t* data_count);
```

**device_read_request**

Asynchronous Function form — Asynchronously read data

```c
kern_return_t device_read_request
    (mach_port_t device,
     mach_port_t reply_port,
     dev_mode_t mode,
     recnum_t recnum,
     int bytes_wanted);
```

**ds_device_read_reply**

Asynchronous Server Interface form — Receive the reply from an asynchronous read

```c
kern_return_t ds_device_read_reply
    (mach_port_t reply_port,
     kern_return_t return_code,
     io_buf_ptr_t data,
     mach_msg_type_number_t data_count);
```

**DESCRIPTION**

The **device_read** function reads a sequence of bytes from a device object. The meaning of `recnum` as well as the specific operation performed is device dependent.

**SECURITY**

The requesting task must hold `dsv_read_device` permission to `device`. 
**device_read**

**PARAMETERS**

- **device**
  [in device port] A device port to the device to be read.

- **reply_port**
  [in reply port] The port to which the reply message is to be sent.

- **mode**
  [in scalar] I/O mode value. Meaningful options are:
  - D_NOWAIT
    Do not wait if data is unavailable.

- **recnum**
  [in scalar] Record number to be read.

- **bytes_wanted**
  [in scalar] Size of data transfer.

- **return_code**
  [in scalar] The return status code from the read.

- **data**
  [out pointer to dynamic array of bytes] Returned data bytes.

- **data_count**
  [out scalar] Number of returned data bytes.

**RETURN VALUE**

`device_read_request` returns only message transmission errors. A return value supplied to `ds_device_read_reply` other than KERN_SUCCESS or MIG_NO_REPLY will cause `mach_msg_server` to de-allocate the returned data. The `return_code` returned by `ds_device_read_reply` or the error return from `device_read` is one of the following:

- **D DEVICE DOWN**
  Device has been shut down.

- **D INVALID RECNUM**
  Invalid record (block) number.

- **D INVALID SIZE**
  Invalid IO size.

- **D IO ERROR**
  Hardware IO error.
Kernel Device Interface

D_NO_SUCH_DEVICE
   No device with that name, or the device is not operational.

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

D_WOULD_BLOCK
   Operation would block, but D_NOWAIT set.

RELATED INFORMATION
   Functions: device_read_inband, device_reply_server.
**device_read_inband**

**Function** — Read a sequence of bytes “inband” from a device object.

**LIBRARY**

```
#include <device/device.h> (device_read_inband)
#include <device/device_request.h> (device_read_request_inband)
#include <device/device_reply.h> (ds_device_read_reply_inband)
```

**SYNOPSIS**

```
kern_return_t device_read_inband
    (mach_port_t device, dev_mode_t mode, recnum_t recnum, int bytes_wanted, io_buf_ptr_inband_t* data, mach_msg_type_number_t* data_count);
```

**device_read_request_inband**

**Asynchronous Function** form — Asynchronously read data

```
kern_return_t device_read_request_inband
    (mach_port_t device, mach_port_t reply_port, dev_mode_t mode, recnum_t recnum, int bytes_wanted);
```

**ds_device_read_reply_inband**

**Asynchronous Server Interface** form — Receive the reply from an asynchronous read

```
kern_return_t ds_device_read_reply_inband
    (mach_port_t reply_port, kern_return_t return_code, io_buf_ptr_inband_t data, mach_msg_type_number_t data_count);
```

**DESCRIPTION**

The **device_read** function reads a sequence of bytes from a device object. The meaning of **recnum** as well as the specific operation performed is device dependent. This call differs from **device_read** in that the returned bytes are returned “inband” in the reply IPC message.
SECURITY

The requesting task must hold *dsv_read_device* permission to *device*.

PARAMETERS

*device*

[in device port] A device port to the device to be read.

*reply_port*

[in reply port] The port to which the reply message is to be sent.

*mode*

[in scalar] I/O mode value. Meaningful options are:

- `D_NOWAIT`
  
  Do not wait if data is unavailable.

*recnum*

[in scalar] Record number to be read.

*bytes_wanted*

[in scalar] Size of data transfer.

*return_code*

[in scalar] The return status code from the read.

*data*

[out array of bytes] Returned data bytes.

*data_count*

[out scalar] Number of returned data bytes.

RETURN VALUE

*device_read_request_inband* returns only message transmission errors. The return value supplied to *ds_device_read_reply_inband* is irrelevant. The *return_code* returned by *ds_device_read_reply_inband* or the error return from *device_read_inband* is one of the following:

- `DDEVICE_DOWN`
  
  Device has been shut down

- `D_INVALID_RECNUM`
  
  Invalid record (block) number

- `D_INVALID_SIZE`
  
  Invalid IO size
D_IO_ERROR
   Hardware IO error

D_NO_SUCHDEVICE
   No device with that name, or the device is not operational.

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

D_WOULD_BLOCK
   Operation would block, but D_NOWAIT set

RELATED INFORMATION
   Functions: device_read, device_reply_server.
**device_set_filter**

**Function** — Names an input filter for a device

**LIBRARY**

```c
#include <device/device.h>
#include <device/net_status.h>
```

**SYNOPSIS**

```c
kern_return_t device_set_filter
    (mach_port_t device,
     mach_port_t receive_port,
     mach_msg_type_name_t receive_port_type,
     int priority,
     filter_array_t filter,
     mach_msg_type_number_t filter_count);
```

**DESCRIPTION**

The `device_set_filter` function provides a means by which selected data appearing at a device interface can be selected and routed to a port.

The filter command list consists of an array of up to NET_MAX_FILTER (16-bit) values to be applied to incoming messages to determine if those messages should be given to a particular input filter.

Each filter command list specifies a sequences of actions which leave a boolean value on the top of an internal stack. Each 16-bit value of the command list specifies a data (push) operation (high order NETF_NBPO bits) as well as a binary operator (low order NETF_NBPA bits).

The value to be pushed onto the stack is chosen as follows:

- **NETF_PUSHLIT**
  Use the next 16-bit value of the filter as the value.

- **NETF_PUSHZERO**
  Use 0 as the value.

- **NETF_PUSHWORD+N**
  Use 16-bit value $N$ of the “data” portion of the message as the value.

- **NETF_PUSHHDR+N**
  Use 16-bit value $N$ of the “header” portion of the message as the value.
NETF_PUSHIND
Pops the top 32-bit value from the stack and then uses it as an index to
the 16-bit value of the “data” portion of the message to be used as the
value.

NETF_PUSHHDRIND
Pops the top 32-bit value from the stack and then uses it as an index to
the 16-bit value of the “header” portion of the message to be used as
the value.

NETF_PUSHSTK+N
Use 32-bit value N of the stack (where the top of stack is value 0) as
the value.

NETF_NOPUSH
Don’t push a value.

The unsigned value so chosen is promoted to a 32-bit value before being pushed.

Once a value is pushed (except for the case of NETF_NOPUSH), the top two
32-bit values of the stack are popped and a binary operator applied to them
(with the old top of stack as the second operand). The result of the operator is
pushed on the stack. These operators are:

NETF_NOP
Don’t pop off any values and do no operation.

NETF_EQ
Perform an equal comparison.

NETF_LT
Perform a less than comparison.

NETF_LE
Perform a less than or equal comparison.

NETF_GT
Perform a greater than comparison.

NETF_GE
Perform a greater than or equal comparison.

NETF_AND
Perform a bit-wise boolean AND operation.

NETF_OR
Perform a bit-wise boolean inclusive OR operation.
NETF_XOR
   Perform a bit-wise boolean exclusive OR operation.

NETF_NEQ
   Perform a not equal comparison.

NETF_LSH
   Perform a left shift operation.

NETF_RSH
   Perform a right shift operation.

NETF_ADD
   Perform an addition.

NETF_SUB
   Perform a subtraction.

NETF_COR
   Perform an equal comparison. If the comparison is TRUE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CAND
   Perform an equal comparison. If the comparison is FALSE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CNOR
   Perform a not equal comparison. If the comparison is FALSE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CNAND
   Perform a not equal comparison. If the comparison is TRUE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

The scan of the filter list terminates when the filter list is emptied, or a NETF_C... operation terminates the list. At this time, if the final value of the top of the stack is TRUE, then the message is accepted for the filter.

SECURITY
   The requesting task must hold *dsv_set_device_filter* permission to *device*.

PARAMETERS

   *device*

   [in device port] A device port
**device_set_filter**

receive_port
[in filter port] The port to receive the input data that is selected by the filter.

receive_port_type
[in scalar] IPC type of the send right provided to the device; either MACH_MSG_TYPE_MAKE_SEND, MACH_MSG_TYPE_MOVE_SEND or MACH_MSG_TYPE_COPY_SEND.

priority
[in scalar] Used to order multiple receivers.

filter
[pointer to in array of filter_t] The address of an array of filter values.

filter_count

**RETURN VALUE**

D_DEVICE_DOWN
Device has been shut down

D_INVALID_OPERATION
No filter port was supplied.

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.
Kernel Device Interface

device_set_status

Function — Sets device status.

LIBRARY

#include <device/device.h>

SYNOPSIS

kern_return_t device_set_status
(mach_port_t device,
 int flavor,
 dev_status_t status,
 mach_msg_type_number_t status_count);

DESCRIPTION

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

SECURITY

The requesting task must hold dsv_set_device_status permission to device.

PARAMETERS

device
[in device port] A device port to the device to be manipulated.

flavor
[in scalar] The type of status information to set.

status
[pointer to in array of int] The status information to set.

status_count
[in scalar] The size of the status information.

RETURN VALUE

D_DEVICE_DOWN
Device has been shut down

D_IO_ERROR
Hardware IO error
device_set_status

D_NO_SUCHDEVICE
No device with that name, or the device is not operational.

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

D_READ_ONLY
Data cannot be written to this device.

RELATED INFORMATION
Functions: device_get_status.
device_write

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

**LIBRARY**

```
#include <device/device.h> (device_write)
```

```
#include <device/device_request.h> (device_write_request)
```

```
#include <device/device_reply.h> (ds_device_write_reply)
```

**SYNOPSIS**

```
kern_return_t device_write
  (mach_port_t device, dev_mode_t mode, recnum_t recnum,
   io_buf_ptr_t data, mach_msg_type_number_t data_count,
   int* bytes_written);
```

**device_write_request**

**Asynchronous Function** form — Asynchronously write data

```
kern_return_t device_write_request
  (mach_port_t device, mach_port_t reply_port, dev_mode_t mode,
   recnum_t recnum, io_buf_ptr_t data, mach_msg_type_number_t data_count);
```

**ds_device_write_reply**

**Asynchronous Server Interface** form — Receive the reply from an asynchronous write

```
kern_return_t ds_device_write_reply
  (mach_port_t reply_port, kern_return_t return_code,
   int bytes_written);
```

**DESCRIPTION**

The `device_write` function writes a sequence of bytes to a device object. The meaning of `recnum` as well as the specific operation performed is device dependent.

**SECURITY**

The requesting task must hold `dsv_write_device` permission to `device`. 
PARAMETERS

device
[device port] A device port to the device to be written.

reply_port
[reply port] The port to which the reply message is to be sent.

mode
[scalar] I/O mode value. Meaningful options are:

D_NOWAIT
Do not wait for I/O completion.

recnum
[scalar] Record number to be written.

data
[pointer to array of bytes] Data bytes to be written.

data_count
[scalar] Number of data bytes to be written.

return_code
[scalar] The return status code from the write.

bytes_written
[out scalar] Size of data transfer.

RETURN VALUE

device_write_request returns only message transmission errors. The return value supplied to ds_device_write_reply is irrelevant. The return_code returned by ds_device_write_reply or the error return from device_write is one of the following:

D_DEVICE_DOWN
Device has been shut down

D_INVALID_RECNUM
Invalid record (block) number

D_INVALID_SIZE
Invalid IO size

D_IO_ERROR
Hardware IO error
Kernel Device Interface

D_NO_SUCH DEVICE
   No device with that name, or the device is not operational.

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

D_READ_ONLY
   Data cannot be written to this device.

D_WOULD_BLOCK
   Operation would block, but D_NOWAIT set

RELATED INFORMATION
   Functions: device_write_inband, device_reply_server.
device_write_inband

Function — Write a sequence of bytes “inband” to a device object.

LIBRARY

#include <device/device.h> (device_write_inband)
#include <device/device_request.h> (device_write_request_inband)
#include <device/device_reply.h> (ds_device_write_reply_inband)

SYNOPSIS

kern_return_t device_write_inband
(mach_port_t device, dev_mode_t mode, recnum_t recnum, io_buf_ptr_inband_t data, mach_msg_type_number_t data_count, int* bytes_written);

device_write_request_inband
Asynchronous Function form — Asynchronously write data

kern_return_t device_write_request_inband
(mach_port_t device, mach_port_t reply_port, dev_mode_t mode, recnum_t recnum, io_buf_ptr_inband_t data, mach_msg_type_number_t data_count);

ds_device_write_reply_inband
Asynchronous Server Interface form — Receive the reply from an asynchronous write

kern_return_t ds_device_write_reply_inband
(mach_port_t reply_port, kern_return_t return_code, int bytes_written);

DESCRIPTION

The device_write_inband function writes a sequence of bytes to a device object. The meaning of recnum as well as the specific operation performed is device dependent. This call differs from device_write in that the bytes to be written are sent “inband” in the request IPC message.
SECURITY

The requesting task must hold `dsv_write_device` permission to `device`.

PARAMETERS

`device`

[in device port] A device port to the device to be written.

`reply_port`

[in reply port] The port to which the reply message is to be sent.

`mode`

[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT

Do not wait for I/O completion.

`recnum`

[in scalar] Record number to be written.

`data`

[pointer to in array of bytes] Data bytes to be written.

`data_count`

[in scalar] Number of data bytes to be written.

`return_code`

[in scalar] The return status code from the write.

`bytes_written`

[ out scalar] Size of data transfer.

RETURN VALUE

`device_write_request_inband` returns only message transmission errors. The return value supplied to `ds_device_write_reply_inband` is irrelevant. The `return_code` returned by `ds_device_write_reply_inband` or the error return from `device_write_inband` is one of the following:

DDEVICE_DOWN

Device has been shut down

D_INVALID_RECNUM

Invalid record (block) number

D_INVALID_SIZE

Invalid IO size
D_IO_ERROR
   Hardware IO error

D_NO_SUCH_DEVICE
   No device with that name, or the device is not operational.

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

D_READ_ONLY
   Data cannot be written to this device.

D_WOULD_BLOCK
   Operation would block, but D_NOWAIT set

RELATED INFORMATION
   Functions: device_write, device_reply_server.
evc_wait

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

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

```c
kern_return_t evc_wait
   (unsigned int event);
```

**DESCRIPTION**

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

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

**SECURITY**

No restrictions defined.

**PARAMETERS**

`event`

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

**NOTES**

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

KERN_NO_SPACE
There is already a thread waiting for this event.

RELATED INFORMATION

Functions: device_map.
This chapter discusses the specifics of the interface between the DTOS kernel and the Security Server. Interfaces labeled as **Function** are kernel interfaces, where interfaces labeled as **Server Interface** are interfaces to the security server.
**avc_cache_control, avc_cache_control_trap**

**Function** — provides interface to the kernel access vector cache for flushing and preloading the cache.

**LIBRARY**

```c
#include <mach/mach_interface.h>
#include <sys/security.h>
```

**SYNOPSIS**

```c
kern_return_t avc_cache_control
  (mach_port_t HostName, int ControlWord, int PolicyID, vector_table_t VectorTable, int VectorTableSize, aid_relevance_table_t AidvTable, int AidvTableSize);

kern_return_t avc_cache_control_trap
  (int ControlWord, int PolicyID, vector_table_t VectorTable, int VectorTableSize, aid_relevance_table_t AidvTable, int AidvTableSize);
```

**DESCRIPTION**

The **avc_cache_control** function is called by the Security Server whenever it needs to flush the access vector cache or to load required permissions into the access vector cache. One example is when the Security Server switches policies. The **avc_cache_control_trap** function is a system call version of the **avc_cache_control** function. It is used to circumvent some limitations in the MIG messaging scheme with regards to in-line data of greater than 1 (4k) page in length.

**SECURITY**

The client must hold **flush_permission** permission to the **HostName** port.

**PARAMETERS**

*HostName*  
[in mach_port_t] The host name port.
**ControlWord**

[in int] The control word that describes the operations to be performed by this invocation of `avc_cache_control`. The control word format is defined in `sys/security.h`. It is a bit mask with the following functions:

- **AVC_FLUSH_CACHE**: to flush the avc vector cache.
- **AVC_CLEAR_CACHE**: to remove all cache entries (including wired)
- **AVC_RELOAD_INITIAL_STATE**: reinitialize cache to initial state values.
- **AVC_VECTOR_TABLE**: the vector table is present.
- **AVC_AIDV_TABLE**: the aid relevance table is present.

**PolicyId**

[in int] The new policy ID. The `PolicyId` will be incremented with each flush or clear of the avc cache. It may be used to verify that security computations apply to the current policy.

**VectorTable**

[in vector_table_t] The table that contains an array of pairs with associated access vectors to load into, or flush from, the cache.

**VectorTableSize**

[in int] The size (in int’s) of the `VectorTable`.

**AidvTable**

[in aid_relevance_table_t] The aid relevance table that the kernel will use.

**AidvTableSize**

[in int] The size of the aid relevance table specified by the `AidvTable` parameter.

**RETURN VALUE**

0 - The operation was successful.

1 - The operation was not successful.

**RELATED INFORMATION**

none
extract_aid

Macro—Returns the authentication identifier field of the security identifier.

LIBRARY

#include <sys/security.h>

SYNOPSIS

authentication_id_t extract_aid
(security_id_t sid);

DESCRIPTION

The extract_aid macro returns the authentication identifier field of the security identifier sid.

SECURITY

None.

PARAMETERS

sid


RETURN VALUE

Authentication identifier.

RELATED INFORMATION

Functions: extract_mid, make_sid.
extract_mid

---

**Macro**—Returns the mandatory identifier field of the security identifier.

**LIBRARY**

`#include <sys/security.h>`

**SYNOPSIS**

```c
authentication_id_t extract_mid
    (security_id_t sid);
```

**DESCRIPTION**

The `extract_mid` macro returns the mandatory identifier field of the security identifier `sid`.

**SECURITY**

None.

**PARAMETERS**


**RETURN VALUE**

Mandatory identifier.

**RELATED INFORMATION**

Functions: `extract_aid`, `make_sid`. 
make_sid

Macro—Builds a security identifier using a mandatory identifier and an authentication identifier.

LIBRARY

#include <sys/security.h>

SYNOPSIS

security_id_t make_sid
    (mandatory_id_t mid,
     authentication_id_t aid);

DESCRIPTION

The make_sid macro returns a security identifier whose MID and AID fields have the values given in mid and aid.

SECURITY

None.

PARAMETERS

mid
    [in mandatory_id] The input mandatory identifier.

aid

RETURN VALUE

Security identifier.

RELATED INFORMATION

Functions: extract_mid, extract_aid.
SSI_compute_access_vector

Server Interface—Requests an access vector for a source sid to a target sid

LIBRARY

#include <sys/security.h>

SYNOPSIS

kern_return_t SSI_compute_access_vector
            (mach_port_t SSPort,
             security_id_t SourceSID,
             security_id_t TargetSID,
             int Permission,
             int *RequestID,
             mach_access_vector_data_t *AccessVector,
             mach_access_vector_data_t *CacheControlVector,
             mach_access_vector_data_t *NotificationVector,
             mach_access_vector_data_t *AIDRelevanceVector,
             unsigned int *Timeout,
             unsigned int *PolicyId,
             int *Status);

DESCRIPTION

The SSI_compute_access_vector function is called by a client (possibly the Kernel), when a security fault has occurred. The Security Server uses the provided security identifiers to compute the associated permission information. The request may also be made by any task that has access to the Security Server’s general service port.

The decision logic used to compute the permissions between SourceSID to TargetSID is determined by the system’s specific security policy.

SECURITY

The client must hold the service permission ss_kern_compute_av or ss_gen_compute_av respectively to SSPort depending on whether it is the client or master Security Server port. The Security Server must have krpsv_provide_permission to the reply port of this request.

PARAMETERS

SSPort

[in port] The port from which the Security Server accepts service requests. This is either the client or master Security Server port.
SourceSID
[in security_id_t] The security identifier of the subject which is attempting to make an access.

Target_SID
[in security_id_t] The security identifier of the object to which the access is being made.

Permission
[in int] The permission to be checked.

RequestID
[in/out int *] A request identifier returned by the Security Server. Not used.

AccessVector
[out mach_access_vector_data_t *] The access vector which describes the permissions of the SourceSID <-> TargetSID pair.

CacheControlVector
[out mach_access_vector_data_t *] An access vector describing the way the access vector cache is to be controlled. Each non-zero bit in the CacheControlVector indicates that the corresponding permission bit in the AccessVector can be cached.

NotificationVector
[out mach_access_vector_data_t *] An access vector used to control generation of audit information. Each non-zero bit indicates that whenever the corresponding permission bit in the AccessVector is used, an audit event will be generated.

AIDRelevanceVector
[out mach_access_vector_data_t *] An access vector describing which permission bits require authentication identifier (AID) verification. Each non-zero bit in this vector indicates that the corresponding permission bit in the AccessVector requires cross-AID checks. This output parameter is used by the Kernel to update its internal AID relevance table and keep it consistent with the security policy.

Timeout
[out unsigned int *] The absolute clock value at which time the access vector will expire from the cache.

PolicyId
[out unsigned int *] A number representing the current revision of the security policy in force. This number will increment everytime a load_security_policy, swap_security_server, or avc_cache_control with AVC_FLUSH_CACHE bit set is performed.
SSI_compute_access_vector

Status
[out int *] Used to return status information for a security request. Not used.

RETURN VALUE

Generic errors apply.

RELATED INFORMATION

Functions: sec_access_provided.
**SSI_context_to_mid**

*Server Interface*— Returns the mandatory identifier associated with a security context.

**LIBRARY**

```
#include <sys/security.h>
```

**SYNOPSIS**

```c
kern_return_t SSI_context_to_mid
    (mach_port_t SSPort,
     mach_sec_context_t SecurityContext,
     int SecurityContextLength,
     mandatory_id_t *MID);
```

**DESCRIPTION**

The `SSI_context_to_mid` function is called by a client when there is a need to get the security identifier that is related to a particular security context. Please refer to `SSI_mid_to_context` for a description of the full security context.

**SECURITY**

The client must hold `ss_gen_context_to_sid` permission to `SSPort`.

**PARAMETERS**

- **SSPort**
  [in mach_port_t] The port on which the security server receives requests.

- **SecurityContext**
  [in mach_sec_context_t] The security context to convert. It must be fully specified.

- **SecurityContextLength**
  [in int] The length of the security context in bytes + 1. The maximum value is 256.

- **MID**
  [out mandatory_id_t *] The fully specified mandatory identifier associated with the provided security context.

**RETURN VALUE**

Generic errors apply.
SSI_context_to_mid

RELATED INFORMATION

Functions: SSI_short_context_to_mid, SSI_mid_to_context, SSI_mid_to_short_context.
SSI_load_security_policy

Server Interface—Loads the security policy.

LIBRARY

#include <sys/security.h>

SYNOPSIS

kern_return_t SSI_load_security_policy
    (mach_port_t SSPort,
     char * SecurityPolicyDir,
     int NameLength);

DESCRIPTION

The SSI_load_security_policy function loads the security policy found in the
directory SecurityPolicyDir. The database file must be named “database_file”
and the permissions file must be named “permissions_file”. If more than one set
of policies is desired, then files describing the policy must be placed in separate
directories.

SECURITY

The client must hold ss_gen_load_policy permission to SSPort.

PARAMETERS

SSPort
    [in security_id_t] The port on which the security server receives re-
    quests.

SecurityPolicyDir
    [in char *>] The name of the directory which holds the security policy.

NameLength
    [in in] The length of the SecurityPolicyName in bytes + 1. The maxi-
    mum value is 1024.

RETURN VALUE

Generic errors apply.

RELATED INFORMATION

Functions: SSI_transfer_security_server_ports.
SSI_record_name_server

Server Interface—Provides the name server port right to the security server.

LIBRARY

#include <sys/security.h>

SYNOPSIS

void SSI_record_name_server
(mach_port_t SSPort,
(mach_port_t NameServerPort);

DESCRIPTION

The SSI_record_name_server function gives the Security Server access to the
NameServerPort. The Security Server then registers its client port with the
name server. This allows clients to look up the port with the name server using
the “security_server_port” keyword.

SECURITY

The client must hold ss_kern_record_name_server permission to SSPort.

PARAMETERS

SSPort
[in mach_port_t] The port on which the Security Server receives re-
quests.

NameServerPort
[in mach_port_t] The port on which the name server receives requests.

RETURN VALUE

Generic errors apply.

RELATED INFORMATION.

Functions: netname_lookup.
### SSI_register_caching_server

**Server Interface**— Provide a means for programs caching security information to be notified of a flush event.

**LIBRARY**

```c
#include <sys/security.h>
```

**SYNOPSIS**

```c
void SSI_register_caching_server(mach_port_t SSPort, mach_port_t FlushNotificationPort);
```

**DESCRIPTION**

The `SSI_register_caching_server` function provides an interface that may be used by other servers caching security information that wish to be notified of a security cache flush event. The supplied port will receive a message containing the policy ID, upon the security server requesting a flush cache. The message format is defined as follows:

```c
c simpleroutine flush_notify
    (SSPort :mach_port_t;
    PolicyId :int);
```

**SECURITY**

The client must hold `ss_gen_register` permission to the security server client port.

**PARAMETERS**

- **SSPort**
  
  [in mach_port_t] The port on which the Security Server receives client requests.

- **FlushNotificationPort**
  
  [in mach_port_t] The port to which the Security Server sends a message when a flush event occurs.

**RETURN VALUE**

Generic errors apply.
SSI_register_caching_server

RELATED INFORMATION.

None.
Security Server Interface

SSI_short_context_to_mid

Server Interface—Returns the mandatory identifier (MID) associated with a security context specified in the short format.

LIBRARY

#include <sys/security.h>

SYNOPSIS

kern_return_t SSI_short_context_to_mid
  (mach_port_t SSPort,
   mach_sec_context_t SecurityContext,
   int SecurityContextLength,
   mandatory_id_t ParentMID,
   mandatory_id_t *MID);

DESCRIPTION

The SSI_short_context_to_mid function is called by a client when there is a need to get the mandatory identifier that is related to a particular security context. This function differs from SSI_context_to_mid in that it accepts the short format of the security context and returns a MID whose classifier field is unspecified. This allows “smart” servers to manage this field in a consistent manner with the Security Server. Please refer to SSI_mid_to_short_context for a description of the short security context format.

SECURITY

The client must hold ss_gen_context_to_sid permission to SSPort.

PARAMETERS

SSPort
  [in mach_port_t] The port on which the Security Server receives requests.

SecurityContext
  [in mach_sec_context_t] The short security context to convert.

SecurityContextLength
  [in int] The length of the security context in bytes + 1. The maximum value is 256.
SSI_short_context_to_mid

**ParentMID**

[in mandatory_id_t ] If the SecurityContext contains fields that are unspecified, then the corresponding values are inherited from the context associated with the ParentMID.

**MID**

[out mandatory_id_t *] The mandatory identifier associated with the provided security context.

**RETURN VALUE**

Generic errors apply.

**RELATED INFORMATION**

Functions: SSI_context_to_mid, SSI_short_mid_to_context, SSI_mid_to_context.
### SSI_mid_to_context

**Server Interface**—Returns the security context associated with a mandatory identifier.

#### LIBRARY

```c
#include <sys/security.h>
```

#### SYNOPSIS

```c
define kern_return_t SSI_mid_to_context
  (mach_port_t SSPort,
   mandatory_id_t MID,
   mach_sec_context_t *SecurityContext,
   int *SecurityContextLength);
```

#### DESCRIPTION

The **SSI_mid_to_context** function is called by a client when there is a need to get the security context that is related to a particular mandatory identifier. The security context is fully specified following the format “Domain/Type : Level : Categories : Classifier”. The Domain/Type field contains either a domain name or a type name. The Level, Categories and Classifier fields have the level name, the comma separated list of category names and the classifier name, respectively.

#### SECURITY

The client must hold **ss_gen_sid_to_context** permission to **SSPort**.

#### PARAMETERS

- **SSPort**
  - [in mach_port_t] The port on which the Security Server receives requests.

- **MID**
  - [in mandatory_id_t] The mandatory identifier to convert. It must be fully specified.

- **SecurityContext**
  - [out mach_sec_context_t *] The full security context associated with the provided security identifier.

- **SecurityContextLength**
  - [in/out int *] The length of the security context in bytes + 1. On input, the variable has the maximum length that the security context can be
SSI_mid_to_context

(256). On output, it contains the actual length of the security context + 1.

RETURN VALUE

Generic errors apply.

RELATED INFORMATION.

Functions: SSI_context_to_mid, SSI_short_context_to_mid,
SSI_mid_to_short_context.
**SSI_mid_to_short_context**

Server Interface—Returns the short format of the security context associated with a mandatory identifier (MID).

**LIBRARY**

`#include <sys/security.h>`

**SYNOPSIS**

```c
kern_return_t SSI_mid_to_short_context
    (mach_port_t SSPort,
     mandatory_id_t MID,
     mach_sec_context_t *SecurityContext,
     int *SecurityContextLength);
```

**DESCRIPTION**

The `SSI_mid_to_short_context` function is called by a client when there is a need to get the short format of the security context that is related to a particular mandatory identifier. This function differs from `SSI_mid_to_context` in that the classifier field of the `MID` need not be specified. The short security context format is “Domain/Type : Level : Categories” where the “;” is a field separator. The Domain/Type field can contain either a domain name or a type name. The Level and Categories fields have the security level name and a list of comma separated category names, respectively. Note that the short security context differs from the full context in that it does not have a classifier field, therefore, the corresponding field in the `MID` is not necessary.

**SECURITY**

The client must hold `ss_gen_sid_to_context` permission to `SSPort`.

**PARAMETERS**

- **SSPort**
  `[in mach_port_t]` The port on which the Security Server receives requests.

- **MID**
  `[in mandatory_id_t]` The mandatory identifier to convert. The classifier field may be unspecified.

- **SecurityContext**
  `[out mach_sec_context_t *]` The short security context associated with the provided mandatory identifier.
SSI_mid_to_short_context

**SecurityContextLength**

[in/out int *] The length of the security context in bytes + 1. On input, the variable has the maximum length that the security context can be (256). On output, it contains the actual length of the security context + 1.

**RETURN VALUE**

Generic errors apply.

**RELATED INFORMATION.**

Functions: SSI_context_to_mid, SSI_short_context_to_mid, SSI_mid_to_context.
SSI_transfer_security_server_ports

Server Interface—Request to transfer Security Server functions to a new program.

LIBRARY
#include <sys/security.h>

SYNOPSIS
kern_return_t SSI_transfer_security_server_ports
(mach_port_t SSPort,
 mach_port_t * Master_port,
 mach_port_t * Client_port,
 int * Policy_id,
 mach_port_array_t Caching_control_ports[],
 int * Caching_control_port_count,
 mach_opaque_table_t opaque_table[],
 int * opaque_table_count,
 mandatory_id_t last_opaque);

DESCRIPTION
The SSI_transfer_security_server_ports function wrests control of security services from the current security server, and returns the receive rights for the security services to the calling program.

SECURITY
The client must hold ss_gen_transfer permission to SSPort.

PARAMETERS
SSPort
[in mach_port_t] The port on which the Security Server receives requests.

Master_port
[out mach_port_t *] The Security Server master (kernel) port receive right.

Client_port
[out mach_port_t *] The Security Server client port receive right.

Policy_id
[out int *] The current policy ID.
**Caching_control_ports**
[out mach_port_array_t *] Array of send rights to ports representing other servers in the system that need to be notified of cache flush events.

**Caching_control_port_count**
[out int*] The number of `caching_control_ports` passed in the array.

**opaque_table**
[out mach_opaque_table_t *] Array of internal MID to opaque MID translations. This table is used to give the new security server knowledge of the existing opaque MIDs running in the system. The table consists of an array, with each element in the array containing the pairing of an opaque and internal MID.

**opaque_table_count**
[out int *] The number of internal<>opaque translations.

**last_opaque**
[out mandatory_id_t *] The last assigned opaque MID. This is the last opaque MID that had been assigned by the old Security Server.

**RETURN VALUE**
Generic errors apply.

**RELATED INFORMATION.**
Functions: `SSI_load_security_policy`. 
SSI_transition_domain

Server Interface—Returns a subject SID and an object SID based on the transition domain and MLS level of the input object SID and subject SID, respectively.

LIBRARY

#include <sys/security.h>

SYNOPSIS

kern_return_t SSI_transition_domain
    (mach_port_t SSPort,
     security_id_t InSSID,
     security_id_t InOSID,
     security_id_t *OutSSID);

DESCRIPTION

The SSI_transition_domain function is called by a client to obtain a subject SID that is of the appropriate domain and security level. If the security policy provides a rule that associates an object type with a transition domain, then the output subject SID corresponds to a security context that has this transition domain. The remaining security context data is the same as that associated with the input subject SID. If no rule is provided by the security policy, the subject SID returned is the same as the input subject SID.

Used in conjunction with the Unix system call execve() or execve_secure(), this feature allows a client to automatically transition to a new domain as a result of executing a file of a particular type. For example, the file /bin/passwd is labeled with the security context passwdTExec:unclassified:none and /etc/passwd is labeled with passwdTFile:unclassified:none. The security policy database indicates that:

• When a subject executes a file labeled passwdTExec, it will transition to passwdD domain
• Only passwdD subjects can write to passwdTFile files
• passwdD subjects can only access passwdTExec memory
• user subjects cannot write to passwdTExec memory or files

A process labeled with the security context user:unclassified:none invokes execve() on the file /bin/passwd. The Unix server calls SSI_context_to_mid() to convert the input security contexts user:unclassified:none and passwdTExec:unclassified:none into the corresponding subject and object SIDs, respectively. It then calls SSI_transition_domain() on these two SIDs to obtain the new transition SID. Finally, the Unix server starts the
new process labeled with the output subject SID. This new process labeled as 
`passwdD` can write to `/etc/passwd` which is labeled as `passwdTFile`.

Another process labeled as `user:unclassified:none` tries to call `execve_secure()` on `vi` with a context of `passwdTExec:unclassified:none`, attempting to bypass `/bin/passwd` and perform arbitrary edits on `/etc/passwd`. The resulting subject security context is `passwdTExec:unclassified:none`, as expected. However, the executable’s object security context is based on `vi`, which is `user:unclassified:none`. The new process cannot read its own text segment and dies.

**SECURITY**

The client must hold `ss_gen_transition` permission to `SSPort`.

**PARAMETERS**

`SSPort`  
[in mach_port_t] The port on which the Security Server receives requests.

`InSSID`  
[in security_id_t] The input subject security identifier.

`InOSID`  
[in security_id_t] The input object security identifier.

`OutSSID`  
[out security_id_t *] The output subject security identifier.

**RETURN VALUE**

Generic errors apply.

**RELATED INFORMATION.**

None.
APPENDIX A  MIG Server Routines

This appendix describes server message de-multiplexing routines generated by MIG from the kernel interface definitions of use to a server in handling messages sent from the kernel.
**device_reply_server**

**Function** — Handles messages from a kernel device driver

**LIBRARY**
libmach_sa.a, libmach.a

Not declared anywhere.

**SYNOPSIS**

```c
boolean_t device_reply_server
    (mach_msg_header_t* in_msg,
     mach_msg_header_t* out_msg);
```

**DESCRIPTION**

The `device_reply_server` function is the MIG generated server handling function to handle messages from kernel device drivers. Such messages were sent in response to the various `device...request`... calls. It is assumed when using those calls that some task is listening for reply messages on the port named as a reply port to those calls. The `device_reply_server` function performs all necessary argument handling for a kernel message and calls one of the device server functions to interpret the message.

**PARAMETERS**

- `in_msg` [pointer to in structure] The device driver message received from the kernel.
- `out_msg` [out structure] A reply message. No messages from a device driver expect a direct reply, so this field is not used.

**RETURN VALUE**

- **TRUE**
  The message was handled and the appropriate function was called.
- **FALSE**
  The message did not apply to this device handler interface and no other action was taken.
device_reply_server

RELATED INFORMATION

Functions:  
- `ds_device_open_reply`
- `ds_device_write_reply`
- `ds_device_write_reply_inband`
- `ds_device_read_reply`
- `ds_device_read_reply_inband`
**exec_server**

**Function** — Handles kernel messages for an exception handler

**LIBRARY**

libmach_sa.a, libmach.a

Not declared anywhere.

**SYNOPSIS**

```c
boolean_t exec_server
    (mach_msg_header_t* in_msg,
     mach_msg_header_t* out_msg);
```

**DESCRIPTION**

The **exec_server** function is the MIG generated server handling function to handle messages from the kernel relating to the occurrence of an exception in a thread. Such messages are delivered to the exception port set via `thread_set_special_port` or `task_set_special_port`. When an exception occurs in a thread, the thread sends an exception message to its exception port, blocking in the kernel waiting for the receipt of a reply. The **exec_server** function performs all necessary argument handling for this kernel message and calls `catch_exception_raise`, which should handle the exception. If `catch_exception_raise` returns KERN_SUCCESS, a reply message will be sent, allowing the thread to continue from the point of the exception; otherwise, no reply message is sent and `catch_exception_raise` must have dealt with the exception thread directly.

**PARAMETERS**

- **in_msg**
  [pointer to in structure] The exception message received from the kernel.

- **out_msg**
  [out structure] A reply message.

**RETURN VALUE**

- **TRUE**
  The message was handled and the appropriate function was called.

- **FALSE**
  The message did not apply to the exception mechanism and no other action was taken.
RELATED INFORMATION

Functions:
- thread_set_special_port,
- task_set_special_port,
- catch_exception_raise.
memory_object_default_server

Function — Handles kernel messages for the default memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t memory_object_default_server
    (mach_msg_header_t* in_msg,
     mach_msg_header_t* out_msg);

DESCRIPTION

The memory_object_default_server function is the MIG generated server handling function to handle messages from the kernel targeted to the default memory manager. This server function only handles messages unique to the default memory manager. Messages that are common to all memory managers are handled by memory_object_server.

A memory manager is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The memory_object_default_server function performs all necessary argument handling for a kernel message and calls one of the default memory manager functions.

PARAMETERS

in_msg
    [pointer to in structure] The memory manager message received from the kernel.

out_msg
    [out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
    The message was handled and the appropriate function was called.

FALSE
    The message did not apply to this memory management interface and no other action was taken.
RELATED INFORMATION

Functions: seqnos_memory_object_default_server, memory_object_server, 
memory_object_create, memory_object_data_initialize, 
default_pager_info, default_pager_object_create.
memory_object_server

Function — Handles kernel messages for a memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t memory_object_server
(mach_msg_header_t* in_msg,
mach_msg_header_t* out_msg);

DESCRIPTION

The memory_object_server function is the MIG generated server handling function to handle messages from the kernel targeted to a memory manager.

A memory manager is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The memory_object_server function performs all necessary argument handling for a kernel message and calls one of the memory manager functions to interpret the message.

PARAMETERS

in_msg
[pointer to in structure] The memory manager message received from the kernel.

out_msg
[out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE
The message did not apply to this memory management interface and no other action was taken.
RELATED INFORMATION

Functions: memory_object_default_server, memory_object_copy,
memory_object_data_request, memory_object_data_unlock,
memory_object_data_write, memory_object_data_return,
memory_object_init, memory_object_lock_completed,
memory_object_change_completed, memory_object_terminate,
seqnos_memory_object_server.
**notify_server**

**Function** — Handle kernel generated IPC notifications

**LIBRARY**

`libmach.a` only

Not declared anywhere.

**SYNOPSIS**

```c
boolean_t notify_server
  (mach_msg_header_t* in_msg,
   mach_msg_header_t* out_msg);
```

**DESCRIPTION**

The `notify_server` function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a `mach_msg` or `mach_port_request_notification` call. The `notify_server` function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

**PARAMETERS**

- `in_msg` [pointer to in structure] The notification message received from the kernel.

- `out_msg` [out structure] Not used.

**RETURN VALUE**

- **TRUE**
  The message was handled and the appropriate function was called.

- **FALSE**
  The message did not apply to the notification mechanism and no other action was taken.

**RELATED INFORMATION**

Functions:

- `seqnos_notify_server`
- `mach_msg`
- `mach_port_request_notification`
- `do_mach_notify_dead_name`
- `do_mach_notify_msg_accepted`
- `do_mach_notify_no_senders`
notify_server

\begin{verbatim}
do_mach_notify_port_deleted, do_mach_notify_port_destroyed,
do_mach_notify_send_once.
\end{verbatim}
prof_server

Function — Handle kernel generated PC sample messages

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t prof_server (mach_msg_header_t* in_msg,
                       mach_msg_header_t* out_msg);

DESCRIPTION

The prof_server function is the MIG generated server handling function to handle messages from the kernel corresponding to program counter (profiling) samples. Such messages are delivered to the task or thread sample port set by task_sample or thread_sample. The prof_server function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

PARAMETERS

in_msg
[pointer to in structure] The sample message received from the kernel.

out_msg
[out structure] Not used.

RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE
The message did not apply to the sample mechanism and no other action was taken.

RELATED INFORMATION

Functions: receive_samples.
seqnos_memory_object_default_server

Function — Handles kernel messages for the default memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t seqnos_memory_object_default_server
  (mach_msg_header_t* in_msg,
   mach_msg_header_t* out_msg);

DESCRIPTION

The seqnos_memory_object_default_server function is the MIG generated server handling function to handle messages from the kernel targeted to the default memory manager. This server function only handles messages unique to the default memory manager. Messages that are common to all memory managers are handled by seqnos_memory_object_server.

A memory manager is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The seqnos_memory_object_default_server function performs all necessary argument handling for a kernel message and calls one of the default memory manager functions.

PARAMETERS

in_msg
  [pointer to in structure] The memory manager message received from the kernel.

out_msg
  [out structure] A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

NOTES

seqnos_memory_object_default_server differs from memory_object_default_server in that it supplies message sequence numbers to the server interfaces it calls.
RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE
The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION
Functions: memory_object_default_server, seqnos_memory_object_server, seqnos_memory_object_create, seqnos_memory_object_data_initialize, seqnos_default_pager_info, seqnos_default_pager_object_create.
seqnos_memory_object_server

Function  —  Handles kernel messages for a memory manager

LIBRARY

libmach.a only

Not declared anywhere.

SYNOPSIS

boolean_t seqnos_memory_object_server
(mach_msg_header_t*    in_msg,
mach_msg_header_t*    out_msg);

DESCRIPTION

The seqnos_memory_object_server function is the MIG generated server handling function to handle messages from the kernel targeted to a memory manager.

A memory manager is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The seqnos_memory_object_server function performs all necessary argument handling for a kernel message and calls one of the memory manager functions to interpret the message.

PARAMETERS

in_msg
(pointer to in structure) The memory manager message received from the kernel.

out_msg
(out structure) A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

NOTES

seqnos_memory_object_server differs from memory_object_server in that it supplies message sequence numbers to the server interfaces.

RETURN VALUE

TRUE

The message was handled and the appropriate function was called.
FALSE

The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions:
seqnos_memory_object_default_server, seqnos_memory_object_copy, seqnos_memory_object_data_request, seqnos_memory_object_data_unlock, seqnos_memory_object_data_write, seqnos_memory_object_data_return, seqnos_memory_object_data_unlock, seqnos_memory_object_init, seqnos_sequence_memory_object_change_completed, seqnos_memory_object_lock_completed, seqnos_memory_object_terminate, memory_object_server.
**seqnos_notify_server**

**Function** — Handle kernel generated IPC notifications

**LIBRARY**

`libmach.a` only

Not declared anywhere.

**SYNOPSIS**

```c
boolean_t seqnos_notify_server(mach_msg_header_t* in_msg,
                                mach_msg_header_t* out_msg);
```

**DESCRIPTION**

The `seqnos_notify_server` function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a `mach_msg` or `mach_port_request_notification` call. The `seqnos_notify_server` function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller.

**PARAMETERS**

`in_msg`  
[pointer to in structure] The notification message received from the kernel.

`out_msg`  
[out structure] Not used.

**NOTES**

`seqnos_notify_server` differs from `notify_server` in that it supplies message sequence numbers to the server interfaces.

**RETURN VALUE**

`TRUE`  
The message was handled and the appropriate function was called.

`FALSE`  
The message did not apply to the notification mechanism and no other action was taken.
RELATED INFORMATION

Functions: notify_server, mach_msg, mach_port_request_notification,
do_seqnos_mach_notify_dead_name,
do_seqnos_mach_notify_msg_accepted,
do_seqnos_mach_notify_no_senders,
do_seqnos_mach_notify_port_deleted,
do_seqnos_mach_notify_port_destroyed,
do_seqnos_mach_notify_send_once.
In general, the default memory manager is just like any other memory manager, except that it is “trusted” to respond promptly to paging requests in as much as that it is the memory manager of last resort. There are a few special requests issued to the default memory manager having to do with the creation and management of anonymous memory.
**Default Memory Management Interface**

**default_pager_info**

**Server Interface** — Return default partition information

**LIBRARY**

libmach.a only

```c
#include <mach/default_pager_object.h>
```

**SYNOPSIS**

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

```c
kern_return_t seqnos_default_pager_info
(mach_port_t pager,
 mach_port_seqno_t seqno,
 vm_size_t* total,
 vm_size_t* free);
```

**DESCRIPTION**

A `default_pager_info` function is called as the result of a message requesting that the default memory manager return information concerning the default pager’s default paging partition. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding a default memory managed object port.

**PARAMETERS**

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

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

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

- **free**
  [out scalar] Free space in the default partition.
RETURN VALUE

The default memory manager should return KERN_SUCCESS if it returns the desired information and KERN_FAILURE if it does not support the operation.

RELATED INFORMATION

Functions:

vm_set_default_memory_manager,
memory_object_default_server, sequos_memory_object_default_server.
**default_pager_object_create**

**Server Interface** — Create a memory object managed by the default pager

**LIBRARY**

libmach.a only

#include `<mach/default_pager_object.h>`

**SYNOPSIS**

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

```c
seqnos_default_pager_object_create
    Sequence Number form
kern_return_t seqnos_default_pager_object_create
    (mach_port_t pager,
     mach_port_seqno_t seqno,
     memory_object_t* memory_object,
     vm_size_t object_size);
```

**DESCRIPTION**

A **default_pager_object_create** function is called as the result of a message requesting that the default memory manager create and return a (shared) memory object which is suitable for use with **vm_map**. This memory object has the same properties as does a memory object provided by **vm_allocate**: its initial contents are zero and the backing contents are temporary in that they do not persist after the memory object is destroyed. The memory object is suitable for use as non-permanent shared memory. The kernel does not make this call itself (which is why it can be a synchronous call); this request is only issued by (privileged) tasks holding a default memory managed object port. This call should be contrasted with the kernel's **memory_object_create** message, in which the memory cache object is already created and the identity of the abstract memory object is made known to the default manager.

**PARAMETERS**

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

- **seqno**
  [in scalar] The sequence number of this message relative to the pager port.
default_pager_object_create

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

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

RETURN VALUE
   Return KERN_SUCCESS if the object was created.

RELATED INFORMATION
   Functions: vm_map, vm_set_default_memory_manager, memory_object_create, memory_object_default_server, seqnos_memory_object_default_server.
memory_object_create

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

LIBRARY
Not declared anywhere.

SYNOPSIS

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

seqnos_memory_object_create

Sequence Number form

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

DESCRIPTION

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

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

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

PARAMETERS

old_memory_object
[in default-pager port] An existing abstract memory object provided by the default memory manager.

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

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

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

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

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

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

NOTES

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

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

The contents of a kernel-created (as opposed to a user-created) memory object can be modified only in main memory. The default memory manager must not change the contents of a temporary memory object, or allow unrelated tasks to access the memory object, control, or name port.
The kernel provides the size of a temporary memory object based on the allocated size. Since the object is not mapped by other tasks, the object will not grow by explicit action. However, the kernel may coalesce adjacent temporary objects in such a way that this object may appear to grow. As such, the supplied object size is merely a hint as to the maximum size.

**RETURN VALUE**

Any return value other than KERN_SUCCESS or MIG_NO_REPLY causes `mach_msg_server` to remove the abstract memory object, memory cache control and memory cache name port references.

**RELATED INFORMATION**

Functions:  
- `default_pager_object_create`,  
- `memory_object_data_initialize`,  
- `memory_object_data_unavailable`,  
- `memory_object_default_server`,  
- `seqnos_memory_object_default_server`.  

memory_object_data_initialize

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

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

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

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

**DESCRIPTION**

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

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

**PARAMETERS**

- `memory_object`
  
  [in abstract-memory-object port] The abstract memory object port that represents the memory object data, as supplied by the kernel in a `memory_object_create` call.
Default Memory Management Interface

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

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

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

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

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

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

**RELATED INFORMATION**
Functions: `memory_object_create`, `memory_object_data_return`, `memory_object_default_server`, `seqnos_memory_object_default_server`.
vm_set_default_memory_manager

Function — Sets the default memory manager.

SYNOPSIS

```c
kern_return_t vm_set_default_memory_manager
(mach_port_t host_priv,
 mach_port_t* default_manager);
```

DESCRIPTION

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

SECURITY

The requesting task must hold `hpsv_set_default_memory_mgr` permission to `host_priv`.

PARAMETERS

- **host_priv**
  
  [in host-control port] The control port naming the host for which the default memory manager is to be set.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: `memory_object_create`, `vm_allocate`. 
Support for multicomputers is being added to the Mach kernel. This provides transparent support for distributed, non-shared-memory environments. The current support does not handle node failures and so is suitable to multicomputer environments but not yet to networked workstation environments.

With this support, a single logical Mach kernel is formed that spans a set of computers. This support transparently distributes Mach IPC and virtual memory. However, each host (called a node) within the multicomputer maintains its identity (separate control and name ports, processor sets, devices, etc.).

This appendix describes operations that apply to individual nodes in such a configuration.
norma_get_special_port

Function — Returns a send right to a node specific port

LIBRARY

#include <mach/norma_special_ports.h>

SYNOPSIS

kern_return_t norma_get_special_port
  (mach_port_t host_priv,
   int node,
   int which_port,
   mach_port_t* special_port);

norma_get_device_port

Macro form

kern_return_t norma_get_device_port
  (mach_port_t host_priv,
   int node,
   mach_port_t* special_port)

⇒ norma_get_special_port (host_priv, node, NORMA_DEVICE_PORT, special_port)

norma_get_host_paging_port

Macro form

kern_return_t norma_get_host_paging_port
  (mach_port_t host_priv,
   int node,
   mach_port_t* special_port)

⇒ norma_get_special_port (host_priv, node, NORMA_HOST_PAGING_PORT, special_port)

norma_get_host_port

Macro form

kern_return_t norma_get_host_port
  (mach_port_t host_priv,
   int node,
   mach_port_t* special_port)

⇒ norma_get_special_port (host_priv, node, NORMA_HOST_PORT, special_port)

norma_get_host_priv_port

Macro form

kern_return_t norma_get_host_priv_port
  (mach_port_t host_priv,
### norma_get_special_port

```c
int
mach_port_t*
node,
special_port)
⇒ norma_get_special_port (host_priv, node, NORMA_HOST_PRIV_PORT,
special_port)
```

### norma_get_nameserver_port

Macro form

```c
kern_return_t norma_get_nameserver_port
(mach_port_t host_priv,
int node,
mach_port_t* special_port)
⇒ norma_get_special_port (host_priv, node,
NORMA_NAMESERVER_PORT, special_port)
```

### DESCRIPTION

The **norma_get_special_port** function returns a send right for a special port belonging to `node` on `host_priv`.

Each node maintains a (small) set of node specific ports. The device master port, host paging port, host name and host control port are maintained by the kernel. The kernel also permits a small set of server specified node specific ports; the name server port is an example and is given (by convention) an assigned special port index.

### PARAMETERS

- **host_priv**
  - [in host-control port] The control port for the host for which to return the special port’s send right.

- **node**
  - [in scalar] The index of the node for which the port is desired.

- **which_port**
  - [in scalar] The index of the special port for which the send right is requested. Valid values are:
    - NORMADEVICE_PORT
      - [device-master port] The device master port for the node.
    - NORMA_HOST_PAGING_PORT
      - [default-pager port] The default pager port for the node.
NORMA_HOST_PORT
[host-name port] The host name port for the node. If the specified node is the current node, this value (unless otherwise set) is the same as would be returned by \texttt{mach\_host\_self}.

NORMA_HOST_PRIV_PORT
[host-control port] The host control port for the node.

NORMA_NAMESERVER_PORT
[name-server port] The registered name server port for the node.

\textit{special\_port}
[out norma-special port] The returned value for the port.

\textbf{RETURN VALUE}

Only generic errors apply.

\textbf{RELATED INFORMATION}

Functions: \texttt{mach\_host\_self}, \texttt{norma\_set\_special\_port}, \texttt{vm\_set\_default\_memory\_manager}. 
norma_port_location_hint

**Function** — Guess a port’s current location

**LIBRARY**
Not declared anywhere.

**SYNOPSIS**

```c
kern_return_t norma_port_location_hint
    (mach_port_t task,
     mach_port_t port,
     int* node);
```

**DESCRIPTION**

The `norma_port_location_hint` function returns the best guess of `port`'s current location. The hint is guaranteed to be a node where the port once was; it is guaranteed to be accurate if port has never moved. This can be used to determine residence node for hosts, tasks, threads, etc.

**PARAMETERS**

- `task`
  [in task port] Task reference (not currently used)

- `port`
  [in random port] Send right to the port to locate.

- `node`
  [out scalar] Port location hint

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `task_set_child_node`, `norma_task_create`. 
norma_set_special_port

**Function** — Sets a node specific special port

**LIBRARY**

```c
#include <mach/norma_special_ports.h>
```

**SYNOPSIS**

```c
kern_return_t norma_set_special_port
    (mach_port_t host_priv,
     int node,
     int which_port,
     mach_port_t special_port);
```

**norma_set_device_port**

**Macro form**

```c
kern_return_t norma_set_device_port
    (mach_port_t host_priv,
     int node,
     mach_port_t special_port)

⇒ norma_set_special_port (host_priv, node, NORMA_DEVICE_PORT, special_port)
```

**norma_set_host_paging_port**

**Macro form**

```c
kern_return_t norma_set_host_paging_port
    (mach_port_t host_priv,
     int node,
     mach_port_t special_port)

⇒ norma_set_special_port (host_priv, node, NORMA_HOST_PAGING_PORT, special_port)
```

**norma_set_host_port**

**Macro form**

```c
kern_return_t norma_set_host_port
    (mach_port_t host_priv,
     int node,
     mach_port_t special_port)

⇒ norma_set_special_port (host_priv, node, NORMA_HOST_PORT, special_port)
```

**norma_set_host_priv_port**

**Macro form**

```c
kern_return_t norma_set_host_priv_port
    (mach_port_t host_priv,
     int node,
     mach_port_t special_port)
```
**norma_set_special_port**

```c
int mach_port_t node,
mach_port_t special_port)
⇒ norma_set_special_port (host_priv, node, NORMA_HOST_PRIV_PORT,
special_port)
```

**norma_set_nameserver_port**

**Macro form**

```c
kern_return_t norma_set_nameserver_port
(mach_port_t host_priv,
int node,
mach_port_t special_port)
⇒ norma_set_special_port (host_priv, node,
NORMA_NAMESERVER_PORT, special_port)
```

**DESCRIPTION**

The `norma_set_special_port` function sets the special port belonging to `node` on `host_priv`.

Each node maintains a (small) set of node specific ports. The device master port, host paging port, host name and host control port are maintained by the kernel. The kernel also permits a small set of server specified node specific ports; the name server port is an example and is given (by convention) an assigned special port index.

**PARAMETERS**

- **host_priv**
  
  [in host-control port] The host for which to set the special port. Currently, this must be the per-node host control port.

- **node**
  
  [in scalar] The index of the node for which the port is to be set.

- **which_port**
  
  [in scalar] The index of the special port to be set. Valid values are:

  - **NORMA_DEVICE_PORT**
    
    [device-master port] The device master port for the node.

  - **NORMA_HOST_PAGING_PORT**
    
    [default-pager port] The default pager port for the node.

  - **NORMA_HOST_PORT**
    
    [host-name port] The host name port for the node.
Multicomputer Support

NORMA_HOST_PRIV_PORT
[host-control port] The host control port for the node.

NORMA_NAMESERVER_PORT
[name-server port] The registered name server port for the node.

special_port
[in norma-special port] A send right to the new special port.

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: mach_host_self, norma_get_special_port, vm_set_default_memory_manager.
norma_task_clone

**Function** — “Clone” a task on a specified node

**LIBRARY**

Not declared anywhere.

**SYNOPSIS**

```c
kern_return_t norma_task_clone
    (mach_port_t parent_task,
     boolean_t inherit_memory,
     int child_node,
     mach_port_t* child_task);
```

**DESCRIPTION**

The **norma_task_clone** function “clones” a new task from **parent_task** on the specified **node** and returns the name of the new task in **child_task**. The child task acquires shared parts of the parent’s address space (see **vm_inherit**) regardless of the inheritance set for the parent’s memory regions, although the inheritance for the child’s regions will be set to that of the parent’s regions. The child task initially contains no threads.

By way of comparison, tasks created by the standard **task_create** primitive are created on the node last set by **task_set_child_node** (by default the **parent_task**’s node).

Other than being created on a different node, the new task has the same properties as if created by **task_create**.

**PARAMETERS**

- **parent_task**
  [in task port] The port for the task from which to draw the child task’s port rights, resource limits, and address space.

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

- **child_node**
  [in scalar] The node index of the node on which to create the child.

- **child_task**
  [out task port] The kernel-assigned port name for the new task.
NOTES

This call differs from `norma_task_create` in that the inheritance set for the parent’s memory regions is ignored; the child always shares memory with the parent.

This call is intended to support process migration, where the inheritance semantics of `norma_task_create` would break migrated programs that depended upon sharing relationships remaining after migration.

This call is not a true task migration call, in that it does not migrate the port space, threads, and other non-address-space attributes of the task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: `task_create`, `norma_task_create`, `task_set_child_node`. 
**norma_task_create**

**Function** — Create a task on a specified node

**LIBRARY**
Not declared anywhere.

**SYNOPSIS**

```c
kern_return_t norma_task_create
    (mach_port_t parent_task,
     boolean_t inherit_memory,
     int child_node,
     mach_port_t* child_task);
```

**DESCRIPTION**

The `norma_task_create` function creates a new task from `parent_task` on the specified `node` and returns the name of the new task in `child_task`. The child task acquires shared or copied parts of the parent’s address space (see `vm_inherit`). The child task initially contains no threads.

By way of comparison, tasks created by the standard `task_create` primitive are created on the node last set by `task_set_child_node` (by default the `parent_task`’s node).

Other than being created on a different node, the new task has the same properties as if created by `task_create`.

**PARAMETERS**

- `parent_task`
  [in task port] The port for the task from which to draw the child task’s port rights, resource limits, and address space.

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

- `child_node`
  [in scalar] The node index of the node on which to create the child.

- `child_task`
  [out task port] The kernel-assigned port name for the new task.
RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: task_create, norma_task_clone, task_set_child_node

SECURITY

The requesting task must hold hsv_get_host_name permission to the processor’s host name port.
task_set_child_node

Function — Set the node upon which future child tasks will be created

LIBRARY
Not declared anywhere.

SYNOPSIS

kern_return_t task_set_child_node
(mach_port_t task, int child_node);

DESCRIPTION
The task_set_child_node function specifies a node upon which child tasks will be created. This call exists only to allow testing with unmodified servers. Server developers should use norma_task_create instead.

PARAMETERS

 task
 [in task port] The task who’s children are to be affected.

 node
 [in scalar] The index of the node upon which future children should be created.

RETURN VALUE
Only generic errors apply.

RELATED INFORMATION
Functions: norma_task_create, norma_task_clone.
This appendix describes special kernel interfaces to support the special hardware features of the Intel 386 processor and its successors.

Aside from the special functions listed here, the Intel 386 support also includes special thread state “flavors” (See `mach/thread_status.h`).

- i386_THREAD_STATE—Basic machine thread state, except for segment and floating registers.
- i386_REGS_SEGS_STATE—Same as i386_THREAD_STATE but also sets/gets segment registers.
- i386_FLOAT_STATE—Floating point registers.
- i386_V86_ASSIST_STATE—Virtual 8086 interrupt table.

(The i386_ISA_PORT_MAP_STATE flavor shown in `mach/thread_status.h` has been disabled.)

**IO Permission Bitmap**

The 386 supports direct IO instructions. Generally speaking, these instructions are privileged (sensitive to IOPL). Mach, in combination with the processor, allows threads to directly execute these instructions against hardware IO ports for which the thread has permission (those named in its IO permission bitmap). (Note that this is a per-thread property.) The `i386_io_port_add` function enables IO to the port corresponding to the device port supplied to the call. `i386_io_port_remove` disables such IO; `i386_io_port_list` lists the devices to which IO is permitted.

For the sake of supporting the DOS emulator, the kernel supports a special device `iopl`. Access to this device implies access to the speaker, configuration CMOS, game port,
sound blaster, printer and the VGA ports (device \textit{kdue} or \textit{vga}). Attempting to execute an IO instruction against one of these devices when the task holds send rights to the \textit{iopl} device automatically adds these devices to the IO permission bitmap.

**Virtual 8086 Support**

Virtual 8086 mode is supported by Mach, enabled when the EFL\_VM (virtual machine) flag in the thread state→\textit{efl} is set. The various instructions sensitive to IOPL are simulated by the Mach kernel. This includes simulating an interrupt enabled flag and associated instructions.

A virtual 8086 task receives simulated 8086 interrupts by setting an interrupt descriptor table (in task space). This table is set with the \texttt{i386\_V86\_ASSIST\_STATE} status flavor.

\begin{verbatim}
[1] struct i386_v86_assist_state
[2] {
[3]   unsigned int int_table;
[4]   int int_count;
[5] }
[6] #define i386_V86_ASSIST_STATE_COUNT
   (sizeof (struct i386_v86_assist_state)/sizeof(unsigned int))
\end{verbatim}

The \textit{int_table} field points to an interrupt table in task space. The table has \textit{int_count} entries. Each entry of this table has the format shown below.

\begin{verbatim}
[1] struct v86_interrupt_table
[2] {
[3]   unsigned int count;
[5]   unsigned short vec;
[6] }
\end{verbatim}

When the 8086 task has an associated interrupt table and its simulated interrupt enable flag is set, the kernel will scan the table looking for an entry whose \textit{count} is greater than zero and whose \textit{mask} value is not set. If found, the count will be decremented and the task will take a simulated 8086 interrupt to the address given by \textit{vec}. No other simulated interrupts will be generated until the 8086 task executes an \texttt{iret} instruction and the (simulated) interrupt enable flag is again set. The generation of the simulated interrupt will turn off the hardware’s trace trap flag; executing the \texttt{iret} instruction will restore the trace trap flag.

**Local Descriptor Table**

Although the 386 (and successors) view the address space as segmented, Mach provides each task with a linear address space (32 bits for the Intel family). The various entries in the system global descriptor table (GDT) are used for system use; in general the entries map all of kernel memory. The thread’s local descriptor table (LDT) maps its task space. Segment 2 of this table is used for task code accesses (it permits only read access); segment 3 is used for data accesses (it permits write access, subject to page level protections); both segments, though, map all of the task’s address space. Segment 1 of the table is unused. Segment 0 is used as a call gate for system calls (traps).
Each thread may set entries in its LDT to describe various ranges of its underlying address space. There is no way that this mechanism permits a thread to access any more virtual memory than its address space permits; these LDT segment entries merely provide different views of the address space. A segment may be thought of as an automatically relocated portion of the address space; the beginning of a segment can be referenced as address zero given the appropriately set 386 segment register. These local segment descriptors are manipulated with the `i386_set_ldt` function and examined with the `i386_get_ldt` function.
**i386_get_ldt**

**Function** — Return per-thread segment descriptors

**LIBRARY**

`#include <mach/i386/mach_i386.h>`

**SYNOPSIS**

```c
[1] struct descriptor
[2] {
[3]     unsigned int low_word;
[4]     unsigned int high_word;
[5] };
[6] typedef struct descriptor descriptor_t;
[7] typedef struct descriptor* descriptor_list_t;

kern_return_t i386_get_ldt(mach_port_t thread,
                           int first_selector,
                           int desired_count,
                           descriptor_list_t* desc_list,
                           mach_msg_type_number_t* returned_count);
```

**DESCRIPTION**

The `i386_get_ldt` function returns per-thread segment descriptors from the thread’s local descriptor table (LDT).

**SECURITY**

The requesting task must hold `thsv_get_thread_info` permission to `thread`.

**PARAMETERS**

- **thread**
  - [in thread port] Thread whose segment descriptors are to be returned

- **first_selector**
  - [in scalar] Selector value (segment register value) corresponding to the first segment whose descriptor is to be returned

- **desired_count**
  - [in scalar] Number of returned descriptors desired
i386_get_ldt

`desc_list`

[unbounded out in-line array of `descriptor_t`] Array of segment descriptors. The reserved size of this array is supplied as the input value for `returned_count`.

`returned_count`

[pointer to in/out scalar] On input, the reserved size of the descriptor array; on output, the number of descriptors returned

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: `i386_set_ldt`. 
**i386_io_port_add**

**Function** — Permit IO instructions to be performed against a device

**LIBRARY**

```
#include <mach/i386/mach_i386.h>
```

**SYNOPSIS**

```c
kern_return_t i386_io_port_add
    (mach_port_t thread, 
mach_port_t device);
```

**DESCRIPTION**

The `i386_io_port_add` function adds a device to the IO permission bitmap for a thread, thereby permitting the thread to execute IO instructions against the device.

**SECURITY**

The requesting task must hold `thsv_set_thread_environment` permission to `thread`.

**PARAMETERS**

- **thread**
  [in thread port] Thread whose permission bitmap is to be set.

- **device**
  [in device port] The device to which IO instructions are to be permitted.

**NOTES**

Normally, the thread must have called `i386_io_port_add` for all devices to which it will execute IO instructions. However, possessing send rights to the `iopl` device port will cause the `iopl` device to be automatically added to the thread’s IO map upon first attempted access. This is a backward compatibility feature for the DOS emulator.

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `i386_io_port_list`, `i386_io_port_remove`. 
i386_io_port_list

Function — List devices permitting IO

LIBRARY

#include <mach/i386/mach_i386.h>

SYNOPSIS

kern_return_t i386_io_port_list
(mach_port_t thread,
device_list_t* list,
mach_msg_type_number_t* count);

DESCRIPTION

The i386_io_port_list function returns a list of the devices named in the thread’s IO permission bitmap, namely those permitting IO instructions to be executed against them.

SECURITY

The requesting task must hold thsv_get_thread_info permission to thread.

PARAMETERS

thread
[in thread port] Thread whose permission list is to be returned

list
[out pointer to dynamic array of device ports] Device ports permitting IO

count
[out scalar] The number of ports returned

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: i386_io_port_add, i386_io_port_remove.
i386_io_port_remove

**Function** — Disable IO instructions against a device

**LIBRARY**

```c
#include <mach/i386/mach_i386.h>
```

**SYNOPSIS**

```c
kern_return_t i386_io_port_remove
    (mach_port_t thread,
     mach_port_t device);
```

**DESCRIPTION**

The `i386_io_port_remove` function removes the specified device from the thread’s IO permission bitmap, thereby prohibiting IO instructions being executed against the device.

**SECURITY**

The requesting task must hold `thsv_set_thread_environment` permission to `thread`.

**PARAMETERS**

- **thread**
  [in thread port] Thread whose permission bitmap is to be cleared

- **device**
  [in device port] Device whose permission is to be revoked

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `i386_io_port_add`, `i386_io_port_list`. 
i386_set_ldt

**Function** — Set per-thread segment descriptors

**LIBRARY**

```
#include <mach/i386/mach_i386.h>
```

**SYNOPSIS**

```
struct descriptor {
    unsigned int low_word;
    unsigned int high_word;
};
typedef struct descriptor descriptor_t;
typedef struct descriptor* descriptor_list_t;

kern_return_t i386_set_ldt(mach_port_t thread, int first_selector, descriptor_list_t desc_list, mach_msg_type_number_t count);
```

**DESCRIPTION**

The `i386_set_ldt` function allows a thread to have a private local descriptor table (LDT) which allows its local segments to map various ranges of its address space.

**SECURITY**

The requesting task must hold `thsv_set_thread_environment` permission to `thread`.

**PARAMETERS**

- `thread` [in thread port] Thread whose segment descriptors are to be set
- `first_selector` [in scalar] Selector value (segment register value) corresponding to the first segment whose descriptor is to be set
- `desc_list` [pointer to in array of `descriptor_t`] Array of segment descriptors. The following forms are permitted:
• Empty descriptor. The ACC_P flag (segment present) may or may not be set.
• ACC_CALL_GATE — Converted into a system call gate. The ACC_P flag must be set.

All other descriptors must have both the ACC_P flag set and specify user mode access (ACC_PL_U).
• ACC_DATA
• ACC_DATA_W
• ACC_DATA_E
• ACC_DATA_EW
• ACC_CODE
• ACC_CODE_R
• ACC_CODE_C
• ACC_CODE_CR
• ACC_CALL_GATE_16
• ACC_CALL_GATE

\[\text{count}\]
[in scalar] Number of descriptors to be set

**RETURN VALUE**

Only generic errors apply.

**RELATED INFORMATION**

Functions: `i386_get_ldt`. 
This appendix discusses the specifics of the various structures used as a part of the kernel’s various interfaces. This appendix does not discuss all of the various data types used by the kernel’s interfaces, only the fields of the various structures used.
host_basic_info

Structure — Defines basic information about a host

SYNOPSIS

```c
[1] struct host_basic_info
[2] {
[3]     int max_cpus;
[4]     int avail_cpus;
[5]     vm_size_t memory_size;
[6]     cpu_type_t cpu_type;
[7]     cpu_subtype_t cpu_subtype;
[8] };
[9] typedef struct host_basic_info host_basic_info_data_t;
[10] typedef struct host_basic_info* host_basic_info_t;
```

DESCRIPTION

The `host_basic_info` structure defines the basic information available about a host.

FIELDS

- `max_cpus`
  Maximum possible CPUs for which kernel is configured

- `avail_cpus`
  Number of CPUs now available

- `memory_size`
  Size of memory, in bytes

- `cpu_type`
  CPU type

- `cpu_subtype`
  CPU sub-type

NOTES

This structure is machine word length specific because of the memory size returned.

RELATED INFORMATION

Functions: `host_info`.

Data structures: `host_load_info`, `host_sched_info`.
host_load_info

Structure — Defines load information about a host

SYNOPSIS

[1] #define CPU_STATE_USER 0
[2] #define CPU_STATE_SYSTEM 1
[3] #define CPU_STATE_IDLE 2
[4] struct host_load_info
[5] { 
[6]     long avenrun[3];
[7]     long mach_factor[3];
[8] };
[9] typedef struct host_load_info host_load_info_data_t;
[10] typedef struct host_load_info* host_load_info_t;

DESCRIPTION

The host_load_info structure defines the loading information available about a host. The information returned is exponential averages over three periods of time: 5, 30 and 60 seconds.

FIELDS

avenrun
load average—average number of runnable processes divided by number of CPUs

mach_factor
The processing resources available to a new thread—the number of CPUs divided by (1 + the number of threads)

RELATED INFORMATION

Functions: host_info.

Data structures: host_basic_info, host_sched_info.
host_sched_info

Structure — Defines scheduling information about a host

SYNOPSIS

```c
[1] struct host_sched_info
[2] {
[3]     int min_timeout;
[4]     int min_quantum;
[5] }
[6] typedef struct host_sched_info host_sched_info_data_t;
[7] typedef struct host_sched_info* host_sched_info_t;
```

DESCRIPTION

The `host_sched_info` structure defines the limiting scheduling information available about a host.

FIELDS

- `min_timeout`  Minimum time-out, in milliseconds
- `min_quantum`  Minimum quantum (period for which a thread can be scheduled if uninterrupted), in milliseconds

RELATED INFORMATION

Functions: `host_info`.

Data structures: `host_basic_info`, `host_load_info`.
mach_access_vector

Structure — Defines the mach access vector which defines the privileges supported by the Mach kernel.

SYNOPSIS

```c
[1] struct mach_access_vector
[2] {
[3]     /* permissions */
[4]         unsigned char av_can_receive: 1,
[5]         av_can_send: 1,
[6]         av_hold_receive: 1,
[7]         av_hold_send: 1,
[8]         av_hold_send_once: 1,
[9]         av_interpose: 1,
[10]        av_specify: 1,
[12]     unsigned char av_transfer_rights: 1,
[13]         av_transfer_send: 1,
[14]         av_transfer_send_once: 1;
[15]     av_transfer_ool: 1,
[16]     mosv_map_vm_region: 1;
[17]     av_set_reply: 1;
[18]     av_unused: 2;
[19]     /* allowed operations */
[20]         union mach_services av_service;
[21]     );
[22] typedef struct mach_access_vector mach_access_vector_data_t;
[23] typedef struct mach_access_vector* mach_access_vector_t;
```

DESCRIPTION

The `mach_access_vector` structure defines the permissions that one security identifier has to another security identifier. The Mach kernel IPC processing is responsible for the enforcement of the permissions upon each attempted use of a port right. In addition the Mach kernel service processing is responsible for the enforcement of the services portion of an access vector before any service is rendered. The general structure provides for 16 permissions and 48 operations. In total it takes two 32 bit words.

FIELDS

- `av_can_receive` Indicates that the task has receive permission to the associated port right.
Data Structures

\textit{av\_can\_send} \\
Indicates that the task has permission to send on the associated port right.

\textit{av\_hold\_receive} \\
Indicates that the task has permission to hold a \texttt{RECEIVE} right.

\textit{av\_hold\_send} \\
Indicates that the task has permission to hold a \texttt{SEND} right.

\textit{av\_hold\_send\_once} \\
Indicates that the task has permission to hold a \texttt{SEND ONCE} right.

\textit{av\_interpose} \\
Indicates that the task has permission to receive messages that were to be received by another security identifier.

\textit{av\_specify} \\
Indicates that the task has permission to specify which security identity is to be associated with a message.

\textit{av\_transfer\_receive} \\
Indicates that the task has permission to transfer a \texttt{RECEIVE} right.

\textit{av\_transfer\_send} \\
Indicates that the task has permission to transfer a \texttt{SEND} right.

\textit{av\_transfer\_send\_once} \\
Indicates that the task has permission to transfer a \texttt{SEND ONCE} right.

\textit{av\_transfer\_ool} \\
Indicates that the task has permission to transfer out-of-line data in a message to the target port.

\textit{mosv\_map\_vm\_region} \\
Controls \texttt{default\_pager\_object\_create, vm\_allocate, vm\_allocate\_secure, vm\_map}.

\textit{av\_service} \\
Defines the services that security policy allows the message receiver to do for the message’s sender. The kernel interprets this portion of the access vector if and if only the kernel is the receiver of the message.

\textbf{NOTES} \\
The contents of an access vector are computed by the Security Server in agreement with a specific security policy and provided to the kernel via interaction with the Security Server. The kernel may cache the access vectors to increase
performance. The kernel provides entries to ensure that the cached vectors may be invalidated.

Functions: `mach_msg_secure`.

Data Structures: `mach_services_t`. 
**mach_device_services**

**Structure** — Defines the services that a task is allowed to request of a device on a kernel device port.

**SYNOPSIS**

```c
[1] struct mach_device_services
[2] {
[3]     unsigned char dsv_close_device: 1,
[4]     dsv_get_device_status: 1,
[5]     dsv_map_device: 1,
[6]     dsv_open_device: 1,
[7]     dsv_read_device: 1,
[8]     dsv_set_device_filter: 1,
[9]     dsv_set_device_status: 1,
[10]    dsv_write_device: 1;
[11]     unsigned char dsv_pager_ctrl: 1,
[12]     dsv_pad: 7
[13]  );
[14] typedef struct mach_device_services mach_device_services_data_t;
[15] typedef struct mach_device_services* mach_device_services_t;
```

**DESCRIPTION**

The `mach_device_services` structure defines the services that a requesting task is allowed to make to a kernel device port.

**SECURITY**

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each device directed kernel request.

**FIELDS**

A **TRUE** value in a specific field indicates that requesting task is allowed to make the request identified in the field.

**RELATED INFORMATION**

Functions: See list above.

Data Structures: `mach_access_vector_t`, `mach_services_t`. 
mach-generic-services

**mach-generic-services**

*Structure* — General data structure to set the maximum size of an allowed operations vector.

**SYNOPSIS**

```c
[1] struct mach_generic_services
[2] {
[3]     unsigned char ago_first_8_bits;
[4]     unsigned char ago_second_8_bits;
[7]     unsigned char ago_fifth_8_bits;
[8]     unsigned char ago_sixth_8_bits;
[9] }
[10] typedef struct mach_generic_services mach_generic_services_data_t;
[11] typedef struct mach_generic_services* mach_generic_services_t;
```

**DESCRIPTION**

The **mach-generic-services** structure established the maximum size of the service vectors. This must be taken into consideration when defining the security database for any system built on the DTOS kernel.

**SECURITY**

Not Applicable.

**FIELDS**

The fields of instances of allowed operations vectors are specified by the system security policy.

**RELATED INFORMATION**

Functions:

Data Structures: **mach-access-vector-t**, and **mach-services-t**.
mach_kernel_reply_port_services

**Structure** — Defines the services that a task is allowed to request of a kernel host server on a kernel host privilege port.

**SYNOPSIS**

```
[1] struct mach_kernel_reply_port_services
[2] {
[3]     unsigned char          krpsv_provide_permission: 1,
[5] }
[6] typedef struct mach_host_priv_services
[7]     mach_kernel_reply_port_services;
[8] typedef struct mach_host_priv_services*
[9]     mach_kernel_reply_port_services_t;
```

**DESCRIPTION**

The `mach_kernel_reply_port_services` structure defines the services that a requesting task is allowed to make on a kernel reply_port.

**SECURITY**

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each reply port that it provides to an external server as a result of a kernel outcall request. The following list indicates which kernel entries are controlled by each service bit.

*krpsv_provide_permission*

Controls reply to `sec_access_provided` service request

**FIELDS**

A `TRUE` value in a specific field indicates that the requesting task is allowed to make the request identified in the field.

**RELATED INFORMATION**

Functions: See list above.

Data Structures: `mach_access_vector_t` and `mach_services_t`. 
mach_host_priv_services

Structure — Defines the services that a task is allowed to request of a kernel host server on a kernel host privilege port.

SYNOPSIS

```c
[1] struct mach_host_priv_services
[2] {
[3]     unsigned char hpsv_get_boot_info: 1,
[4]     hpsv_get_host_processors: 1,
[5]     hpsv_pset_ctrl_port: 1,
[6]     hpsv_reboot_host: 1,
[7]     hpsv_set_default_memory_mgr: 1,
[8]     hpsv_set_time: 1;
[9]     hpsv_wire_thread: 1,
[10]    hpsv_wire_vm: 1;
```

```c
[12] typedef struct mach_host_priv_services mach_host_priv_services_data_t;
[13] typedef struct mach_host_priv_services* mach_host_priv_services_t;
```

DESCRIPTION

The mach_host_priv_services structure defines the services that a requesting task is allowed to make on a kernel host control port.

SECURITY

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each host privileged port directed kernel request.

FIELDS

A TRUE value in a specific field indicates that the requesting task is allowed to make the request identified in the field.

RELATED INFORMATION

Functions: See list above.

Data Structures: mach_access_vector_t and mach_services_t.
mach_host_services

**Structure** — Defines the services that a task is allowed to request of a kernel host server on a kernel host name port.

**SYNOPSIS**

```c
[1] struct mach_host_services
[2] {
[3]   unsigned char hsv_create_pset: 1,
[4]   hsv_flush_permission: 1,
[5]   hsv_get_default_pset_name: 1,
[6]   hsv_get_host_info: 1,
[7]   hsv_get_host_name: 1,
[8]   hsv_get_host_version: 1,
[9]   hsv_get_time: 1,
[10]  hsv_pset_names: 1,
[11]  unsigned char hsv_get_audit_port: 1,
[12]  hsv_get_security_client_port: 1,
[13]  hsv_get_security_master_port: 1,
[14]  hsv_get_special_port: 1,
[15]  hsv_set_audit_port: 1,
[16]  hsv_set_security_client_port: 1,
[17]  hsv_set_security_master_port: 1,
[18]  hsv_set_special_port: 1,
[19]  unsigned char hsv_get_crypto_port: 1,
[20]  hsv_get_host_control_port: 1,
[21]  hsv_get_negotiation_port: 1,
[22]  hsv_get_crypto_port: 1,
[23]  hsv_set_negotiation_port: 1,
[24]  hsv_set_crypto_port: 1,
[25]  hsv_set_negotiation_port: 1,
[26]  hsv_get_network_ss_port: 1;
[27]  unsigned char hsv_set_network_ss_port: 1,
[28]  hsv_pad: 7;
[29] }
[30] typedef struct mach_host_services mach_host_services_data_t;
[31] typedef struct mach_host_services* mach_host_services_t;
```

**DESCRIPTION**

The **mach_host_services** structure defines the services that a requesting task is allowed to make on a kernel host name port.

**SECURITY**

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each host name port directed kernel request.
FIELDS

A TRUE value in a specific field indicates that the requesting task is allowed to
make the request identified in the field.

RELATED INFORMATION

Functions: See list above.

Data Structures: mach_access_vector_t and mach_services_t.
**mach_mem_obj_services**

*Structure* — Defines the services that a task is allowed to request of a kernel host server on memory object ports.

**SYNOPSIS**

```c
[1] struct mach_mem_obj_services
[2] {
[3]     unsigned char mosv_have_execute: 1,
[4]     mosv_have_read: 1,
[5]     mosv_have_write: 1,
[6]     mosv_unused1: 1,
[7]     mosv_page_vm_region: 1,
[8]     mosv_pad: 3;
[9] }

[10] typedef struct mach_mem_obj_services mach_mem_obj_services_data_t;
[11] typedef struct mach_mem_obj_services* mach_mem_obj_services_t;
```

**DESCRIPTION**

The `mach_mem_obj_services` structure defines the services that a requesting task is allowed to make to a kernel processor port.

**SECURITY**

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each memory object directed kernel request.

**FIELDS**

A **TRUE** value in a specific field indicates that the requesting task is allowed to make the request identified in the field.

**RELATED INFORMATION**

Functions: See list above.

Data Structures: `mach_access_vector_t, mach_services_t.`
mach_mem_ctrl_services

Structure — Defines the services that a task is allowed to request on a kernel host server on a memory control port.

SYNOPSIS

```c
[1] struct mach_mem_ctrl_services
[2] {
[3]    unsigned char mcsv_change_page_locks: 1
[4]    mcsv_destroy_object: 1,
[5]    mcsv_get_attributes: 1,
[6]    mcsv_invoke_lock_request: 1,
[7]    mcsv_make_page_precious: 1,
[8]    mcsv_provide_data: 1,
[9]    mcsv_remove_page: 1,
[10]   mcsv_revoke_ibac: 1;
[11]    unsigned char mcsv_save_page: 1,
[12]   mcsv_set_attributes: 1,
[13]   mcsv_set_ibac_port: 1,
[14]   mcsv_supply_ibac: 1,
[16] }
[17] typedef struct mach_mem_ctrl_services mach_mem_ctrl_services_data_t;
[18] typedef struct mach_mem_ctrl_services* mach_mem_ctrl_services_t;
```

DESCRIPTION

The mach_mem_ctrl_services structure defines the services that a requesting task is allowed to make to a kernel memory control port.

SECURITY

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on memory control port directed kernel request.

FIELDS

A TRUE value in a specific field indicates that the requesting task is allowed to make the request identified in the field.

RELATED INFORMATION

Functions: See list above

Data Structures: mach_access_vector_t, mach_services_t.
mach_msg_header

**Structure** — Defines the header portion for messages

**SYNOPSIS**

```c
typedef struct
{
    mach_msg_bits_t msgh_bits;       // MACH_MSGH_BITS_REMOTE_MASK
    mach_msg_size_t msgh_size;       // MACH_MSGH_BITS_LOCAL_MASK
    mach_port_t msgh_remote_port;    // MACH_MSGH_BITS_REMOTE_MASK
    mach_port_t msgh_local_port;     // MACH_MSGH_BITS_LOCAL_MASK
    mach_port_seqno_t msgh_seqno;    // MACH_MSGH_BITS_REMOTE_MASK
    mach_msg_id_t msgh_id;           // MACH_MSGH_BITS_LOCAL_MASK
} mach_msg_header_t;
```

**DESCRIPTION**

A Mach message consists of a fixed size message header, a `mach_msg_header_t`, followed by zero or more data items. Data items are typed. Each item has a type descriptor followed by the actual data (or an address of the data, for out-of-line memory regions).

There are two forms of type descriptors, a `mach_msg_type_t` and a `mach_msg_type_long_t`. The `mach_msg_type_long_t` type descriptor allows larger values for these fields. The `msgtl_header` field in the long descriptor is only used for its in-line, long-form, and de-allocate bits.

**FIELDS**

*msgh_bits*

This field specifies the following properties of the message:

- **MACH_MSGH_BITS_REMOTE_MASK**
  Encodes `mach_msg_type_name_t` values that specify the port rights in the `msgh_remote_port` field. The value must specify a send or send-once right for the destination of the message.

- **MACH_MSGH_BITS_LOCAL_MASK**
  Encodes `mach_msg_type_name_t` values that specify the port rights in the `msgh_local_port` field. If the value doesn’t specify a send or send-once right for the message’s reply port, it must be zero and `msgh_local_port` must be `MACH_PORT_NULL`.
MACH_MSGH_BITS_COMPLEX
The complex bit must be specified if the message body contains port rights or out-of-line memory regions. If it is not specified, then the message body carries no port rights or memory, no matter what the type descriptors may seem to indicate.

MACH_MSGH_BITS_REMOTE(bits)
This macro returns the appropriate mach_msg_type_name_t values, given a msgh_bits value.

MACH_MSGH_BITS_LOCAL(bits)
This macro returns the appropriate mach_msg_type_name_t values, given a msgh_bits value.

MACH_MSGH_BITS (remote, local)
This macro constructs a value for msgh_bits, given two mach_msg_type_name_t values.

msgh_size
In the header of a received message, this field contains the message's size. The message size, a byte quantity, includes the message header, type descriptors, and in-line data. For out-of-line memory regions, the message size includes the size of the in-line address, not the size of the actual data region. There are no arbitrary limits on the size of a Mach message, the number of data items in a message, or the size of the data items.

msgh_remote_port
When sending, specifies the destination port of the message. The field must carry a legitimate send or send-once right for a port. When received, this field is swapped with msgh_local_port.

msgh_local_port
When sending, specifies an auxiliary port right, which is conventionally used as a reply port by the recipient of the message. The field must carry a send right, a send-once right, MACH_PORT_NULL, or MACH_PORT_DEAD. When received, this field is swapped with msgh_remote_port.

msgh_seqno
The sequence number of this message relative to the port from which it is received. This field is ignored on sent messages.

msgh_id
Not set or read by the mach_msg call. The conventional meaning is to convey an operation or function id.
NOTES

Simple messages are provided to handle in-line data. The sender copies the in-line data into the message structure, and the receiver usually copies it out.

Non-simple messages are provided to handle out-of-line data. Out-of-line data allows for the sending of port information or data blocks that are very large or of variable size. The kernel maps out-of-line data from the address space of the sender to the address space of the receiver. The kernel copies the data only if the sender or receiver subsequently modifies it. This is an example of copy-on-write data sharing.

RELATED INFORMATION

Functions: `mach_msg`, `mach_msg_receive`, `mach_msg_send`.

Data Structures: `mach_msg_type`, `mach_msg_type_long`.
Data Structures

mach_msg_type

Structure — Defines the data descriptor for long data items in messages

SYNOPSIS

```
typedef struct {
    unsigned int  msgt_name: 8,
    msgt_size: 8,
    msgt_number: 12,
    msgt_inline: 1,
    msgt_longform: 1,
    msgt_deallocate: 1,
    msgt_unused: 1;
} mach_msg_type_t;
```

DESCRIPTION

Each data item in a MACH IPC message has a type descriptor, a `mach_msg_type_t` or a `mach_msg_type_long_t`. The `mach_msg_type_long_t` type descriptor allows larger values for these fields.

FIELDS

`msgt_name`

Specifies the data's type. The following types are predefined:

- **MACH_MSG_TYPE_UNSTRUCTURED**
  un-interpreted data (32 bits)

- **MACH_MSG_TYPE_BIT**
  single bit

- **MACH_MSG_TYPE_BOOLEAN**
  boolean value (32 bits)

- **MACH_MSG_TYPE_INTEGER_16**
  16 bit integer

- **MACH_MSG_TYPE_INTEGER_32**
  32 bit integer

- **MACH_MSG_TYPE_CHAR**
  single character

- **MACH_MSG_TYPE_BYTE**
  8-bit byte
MACH_MSG_TYPE_INTEGER_8
8-bit integer

MACH_MSG_TYPE_REAL
floating value (32 bits)

MACH_MSG_TYPE_STRING
null terminated

MACH_MSG_TYPE_STRING_C
null terminated

MACH_MSG_TYPE_PORT_NAME
the type of mach_port_t. This is the type of the name for a port, not the type to specify if a port right is to be specified.

MACH_MSG_TYPE_MOVE_RECEIVE
move the name receive right

MACH_MSG_TYPE_MOVE_SEND
move the named send right

MACH_MSG_TYPE_MOVE_SEND_ONCE
move the named send-once right

MACH_MSG_TYPE_COPY_SEND
make a copy of the named send right

MACH_MSG_TYPE_MAKE_SEND
make a send right from the named receive right

MACH_MSG_TYPE_MAKE_SEND_ONCE
make a send-once right from the named send or receive right

The last six types specify port rights, and receive special treatment. The type MACH_MSG_TYPE_PORT_NAME describes port right names, when no rights are being transferred, but just names. For this purpose, it should be used in preference to MACH_MSG_TYPE_INTEGER_32.

\textit{msgt\_size}

Specifies the size of each datum, in bits. For example, the \textit{msgt\_size} of MACH_MSG_TYPE_INTEGER_32 data is 32.

\textit{msgt\_number}

Specifies how many data elements comprise the data item. Zero is a legitimate number. The total length specified by a type descriptor is \((\textit{msgt\_size} * \textit{msgt\_number})\), rounded up to an integral number of bytes. In-line data is then padded to an integral number of long-words.
Data Structures

This ensures that type descriptors always start on long-word boundaries. It implies that message sizes are always an integral multiple of a long-word’s size.

msgt_inline
When FALSE, specifies that the data actually resides in an out-of-line region. The address of the data region follows the type descriptor in the message body. The msgt_name, msgt_size, and msgt_number fields describe the data region, not the address.

msgt_longform
Specifies, when TRUE, that this type descriptor is a mach_msg_type_long_t instead of a mach_msg_type_t.

msgt_deallocate
Used with out-of-line regions. When TRUE, it specifies the data region should be de-allocated from the sender’s address space (as if with vm_deallocate) when the message is sent.

msgt_unused
Not used, should be zero.

RELATED INFORMATION
Functions: mach_msg, mach_msg_receive, mach_msg_send.

Data Structures: mach_msg_header, mach_msg_type_long.
mach_msg_type_long

 Structure — Defines the data descriptor for long data items in messages

SYNOPSIS

```c
[1] typedef struct
[2] {
[3]     mach_msg_type_t msgtl_header;
[4]     unsigned short msgtl_name;
[5]     unsigned short msgtl_size;
[6]     unsigned int msgtl_number;
[7] } mach_msg_type_long_t;
```

DESCRIPTION

Each data item has a type descriptor, a `mach_msg_type_t` or a `mach_msg_type_long_t`. The `mach_msg_type_long_t` type descriptor allows larger values for these fields. The `msgtl_header` field in the long descriptor is only used for its in-line, long-form, and de-allocate bits.

FIELDS

`msgtl_header`

A header in common with `mach_msg_type_t`. When the `msgt_longform` bit in the header is TRUE, this type descriptor is a `mach_msg_type_long_t` instead of a `mach_msg_type_t`. The `msgt_name`, `msgt_size`, and `msgt_number` fields should be zero. Instead, `mach_msg` uses the following: `msgtl_name`, `msgtl_size`, and `msgtl_number` fields.

`msgtl_name`

Specifies the data's type. The defined values are the same as those for `mach_msg_type`.

`msgtl_size`

Specifies the size of each datum, in bits. For example, the `msgtl_size` of MACH_MSG_TYPE_INTEGER_32 data is 32.

`msgtl_number`

Specifies how many data elements comprise the data item. Zero is a legitimate number. The total length specified by a type descriptor is `(msgtl_size * msgtl_number)`, rounded up to an integral number of bytes. In-line data is then padded to an integral number of long-words. This ensures that type descriptors always start on long-word boundaries. It implies that message sizes are always an integral multiple of a long-word’s size.
RELATED INFORMATION

Functions: mach_msg, mach_msg_receive, mach_msg_send.

Data Structures: mach_msg_header, mach_msg_type.
mach_port_status

Structure — Defines information for a port

SYNOPSIS

```
[1] struct mach_port_status
[2] {
[3]     mach_port_t mps_pset;
[4]     mach_port_seqno_t mps_seqno;
[5]     mach_port_mscount_t mps_mscount;
[6]     mach_portMSGcount_t mps_qlimit;
[7]     mach_port_msgcount_t mps_msgcount;
[8]     mach_port_rights_t mps_sorights;
[9]     boolean_t mps_srights;
[10]    boolean_t mps_pdrequest;
[11]    boolean_t mps_nsrequest;
[12]  }
[13]  typedef struct mach_port_status mach_port_status_t;
```

DESCRIPTION

The mach_port_status structure defines information about a port.

FIELDS

mps_pset
    Containing port set

mps_seqno
    Current sequence number for the port.

mps_mscount
    Make-send count

mps_qlimit
    Queue limit

mps_msgcount
    Number in the queue

mps_sorights
    How many send-once rights

mps_srights
    True if send rights exist
Data Structures

mps_pdrequest
True if there is a port-deleted requested

mps_nsrequest
True if no-senders requested

RELATED INFORMATION
Functions: mach_port_get_receive_status.
**mach_proc_services**

**Structure** — Defines the services that a task is allowed to request of a kernel host server on a kernel processor port.

**SYNOPSIS**

```c
[1] struct mach_proc_services
[2] {
[3]   unsigned char psv_assign_processor_to_set: 1,
[4]   psv_get_processor_assignment: 1,
[5]   psv_get_processor_info: 1,
[6]   psv_may_control_processor: 1,
[7]   psv_pad: 4;
[8] }
```

```c
[9] typedef struct mach_proc_services mach_proc_services_data_t;
[10] typedef struct mach_proc_services* mach_proc_services_t;
```

**DESCRIPTION**

The `mach_proc_services` structure defines the services that a requesting task is allowed to make to a kernel processor self port.

**SECURITY**

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each processor port directed kernel request.

**FIELDS**

A **true** value in a specific field indicates that the requesting task is allowed to make the request identified in the field.

**RELATED INFORMATION**

Functions: See list above.

Data Structures: `mach_access_vector_t`, and `mach_services_t`. 
**mach_proc_set_services**

**Structure** — Defines the services that a task is allowed to request of a kernel host server on a kernel processor set port.

**SYNOPSIS**

```c
[1] struct mach_proc_set_services
[2] {
[3]     unsigned char pssv_assign_processor: 1,
[4]     pssv_assign_task: 1,
[5]     pssv_assign_thread: 1,
[6]     pssv_chg_pset_max_pri: 1,
[7]     pssv_define_new_scheduling_policy: 1,
[8]     pssv_destroy_pset: 1,
[9]     pssv_get_pset_info: 1,
[10]    pssv_invalidate_scheduling_policy: 1,
[11]    pssv_observe_pset_processes: 1,
[12]    psv_pad: 5;
[13] };
[14] typedef struct mach_proc_set_services mach_proc_set_services_data_t;
[15] typedef struct mach_proc_set_services* mach_proc_set_services_t;
```

**DESCRIPTION**

The **mach_proc_set_services** structure defines the services that a requesting task is allowed to make to a kernel processor set port.

**SECURITY**

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each processor set directed kernel request.

**FIELDS**

A **TRUE** value in a specific field indicates that the requesting task is allowed to make the request identified in the field.

**RELATED INFORMATION**

Functions: See list above.

Data Structures: **mach_access_vector_t** and **mach_services_t**.
mach_services

Structure — Defines service vectors that control the services that task is allowed to request on kernel ports.

SYNOPSIS

```c
[1] union mach_services
[2] {
[3]     mach_device_services_data_t dev_sv;
[4]     mach_host_priv_services_data_t host_priv_sv;
[5]     mach_host_services_data_t host_sv;
[6]     mach_mem_obj_services_data_t mem_obj_sv;
[7]     mach_mem_ctrl_services_data_t mem_ctrl_sv;
[8]     mach_proc_services_data_t proc_sv;
[9]     mach_proc_set_services_data_t proc_set_sv;
[10]    mach_task_services_data_t task_sv;
[11]    mach_thread_services_data_t thread_sv;
[12]    mach_generic_services_data_t gen_sv;
[13] };
[14] typedef union mach_services mach_services_data_t;
[15] typedef union mach_services *mach_services_t;
```

DESCRIPTION

The `mach_services` union defines the classes of services that the kernel will enforce as well as the general operation vector which may be used by non kernel system servers. The interpretation of the fields of this vector are specified by the system security policy and enforced by the receiver of the associated port.

SECURITY

The service field in the access vector allows the system security policy to specify which services a specific task may make to a particular port. It is the responsibility of a port’s receiver to enforce the information provided in the allowed operations portion of an access vector. This provides two levels of control over operations. First it is possible to deny a task permission to send a message to a port, and second it is possible to control which services will be allowed.

FIELDS

`dev_sv`

Bit vector indicating which device port directed service requests the requesting task is allowed to make.
Data Structures

host_priv_sv
Bit vector indicating which host priv port directed service requests the requesting task is allowed to make.

host_sv
Bit vector indicating which host port directed service requests the requesting task is allowed to make.

mem_obj_sv
Bit vector indicating which object port directed service requests the requesting task is allowed to make.

mem_ctrl_sv
Bit vector indicating which memory control port directed service requests the requesting task is allowed to make.

proc_sv
Bit vector indicating which processor port directed service requests the requesting task is allowed to make.

proc_set_sv
Bit vector indicating which processor set port directed service requests the requesting task is allowed to make.

task_sv
Bit vector indicating which task port directed service requests the requesting task is allowed to make.

thread_sv
Bit vector indicating which thread port directed service requests the requesting task is allowed to make.

gen_sv
A port specific bit vector indicating which services the requesting task is allowed to make to the port. The purpose of this field is to determine the maximum number of bits in a service vector.

RELATED INFORTION

Functions: None.

Data Structures: mach_device_services, mach_host_priv_services,
mach_host_services, mach_mem_obj_services, mach_mem_ctrl_services,
mach_proc_services, mach_proc_set_services, mach_task_services and
mach_thread_services.
mach_task_services

Structure — Defines the services that a task is allowed to request of a kernel host server on a kernel task port.

SYNOPSIS

```
[1] struct mach_task_services
[2] {
[3]     unsigned char tsv_access_mach_nattribute: 1,
[4]     tsv_add_name: 1,
[5]     tsv_add_thread: 1,
[6]     tsv_add_thread_secure: 1,
[7]     tsv_allocate_vm_region: 1,
[8]     tsv_alter_pns_info: 1,
[9]     tsv_assign_task_to_pset: 1,
[10]    tsv_chg_vm_region_prot: 1;
[11]     unsigned char tsv_chg_task_priority: 1,
[12]    tsv_copy_vm: 1,
[13]    tsv_create_task: 1,
[14]    tsv_create_task_secure: 1,
[15]    tsv_deallocate_vm_region: 1,
[16]    tsv_extract_right: 1,
[17]    tsv_get_emulation: 1,
[18]    tsv_get_task_assignment: 1;
[19]     unsigned char tsv_get_task_boot_port: 1,
[20]    tsv_get_task_exception_port: 1,
[21]    tsv_get_task_info: 1,
[22]    tsv_get_task_kernel_port: 1,
[23]    tsv_get_task_threads: 1,
[24]    tsv_get_vm_region_info: 1,
[25]    tsv_get_vm_statistics: 1,
[26]    tsv_lookup_ports: 1;
[27]     unsigned char tsv_manipulate_port_set: 1,
[28]    tsv_observe_pns_info: 1,
[29]    tsv_port_rename: 1,
[30]    tsv_read_vm_region: 1,
[31]    tsv_register_notification: 1,
[32]    tsv_register_ports: 1,
[33]    tsv_remove_name: 1,
[34]    tsv_resume_task: 1;
[35]     unsigned char tsv_sample_task: 1,
[36]    tsv_set_emulation: 1,
[37]    tsv_set_vm_region_inherit: 1,
[38]    tsv_set_ras: 1,
[39]    tsv_set_task_boot_port: 1,
[40]    tsv_set_task_exception_port: 1,
[41]    tsv_set_task_kernel_port: 1,
[42]    tsv_suspend_task: 1;
```
### Data Structures

```c
[43]  unsigned char  tsv_terminate_task: 1,
[44]  tsv_wire_vm_for_task: 1,
[45]  tsv_write_vm_region: 1,
[46]  tsv_cross_context_create: 1,
[47]  tsv_cross_context_inherit: 1,
[48]  tsv_chg_sid: 1,
[49]  tsv_make_sid: 1,
[50]  tsv_transition_sid: 1,
[51] ];

typedef struct mach_task_services
    mach_task_services_data_t;

typedef struct mach_task_services*
    mach_task_services_t;
```

### DESCRIPTION

The `mach_task_services` structure defines the services that a requesting task is allowed to make on a kernel task port.

### SECURITY

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each task directed kernel request.

### FIELDS

A **TRUE** value in a specific field indicates that the requesting task is allowed to make the request identified in the field.

### RELATED INFORMATION

Functions: See list above.

Data Structures: `mach_access_vector_t` and `mach_services_t`. 
**mach_thread_services**

**Structure** — Defines the services that a task is allowed to request of a kernel host server on a kernel thread port.

**SYNOPSIS**

```c
struct mach_thread_services
{
    unsigned char thsv_abort_thread: 1,
    thsv_abort_thread_depress: 1,
    thsv_assign_thread_to_pser: 1,
    thsv_can_switch: 1,
    thsv_can_switch_pri: 1,
    thsv_depress_pri: 1,
    thsv_get_thread_assignment: 1,
    thsv_get_thread_exception_port: 1;
    unsigned char thsv_get_thread_info: 1,
    thsv_get_thread_kernel_port: 1,
    thsv_initiate_secure: 1,
    thsv_raise_exception: 1,
    thsv_resume_thread: 1,
    thsv_sample_thread: 1,
    thsv_set_max_thread_priority: 1;
    unsigned char thsv_set_thread_exception_port: 1,
    thsv_set_thread_kernel_port: 1,
    thsv_set_thread_policy: 1,
    thsv_set_thread_priority: 1,
    thsv_suspend_thread: 1,
    thsv_switch_thread: 1,
    thsv_terminate_thread: 1;
    unsigned char thsv_wait_evc: 1;
    thsv_wire_thread_into_memory: 1;
    thsv_pad 6;
};
typedef struct mach_thread_services mach_thread_services_data_t;
typedef struct mach_thread_services* mach_thread_services_t;
```

**DESCRIPTION**

The **mach_thread_services** structure defines the services that a requesting task is allowed to make on a kernel thread port.
SECURITY

The system security policy specifies the criteria for setting the fields in this vector. The kernel enforces the allowed operations on each thread port directed kernel request.

FIELDS

A TRUE value in a specific field indicates that the requesting task is allowed to make the request identified in the field.

RELATED INFORMATION

Functions: See list above.

Data Structures: mach_access_vector_t and mach_services_t.
mapped_time_value

Structure — Defines format of kernel maintained time in the mapped clock device

SYNOPSIS

```c
[1] struct mapped_time_value
[2] {  
[3]     long     seconds;
[4]     long     microseconds;
[5]     long     check_seconds;
[6] }
[7] typedef struct mapped_time_value mapped_time_value_t;
```

DESCRIPTION

The `mapped_time_value` structure defines the format of the current-time structure maintained by the kernel and visible by mapping (device_map) the “time” pseudo-device. The data in this structure is updated at every clock interrupt. It contains the same value that would be returned by `host_get_time`.

FIELDS

`seconds`
Seconds since system initialization

`microseconds`
Microseconds in the current second

`check_seconds`
A field used to synchronize with the kernel’s setting of the time.

NOTES

Because of the race between the referencing of these multiple fields and the kernel’s setting them, they should be referenced as follows:

```c
[1] do
[2] {  
[3]     secs = mtime → seconds;
[4]     usecs = mtime → microseconds;
[5] } while (secs!= mtime → check_seconds);
```

RELATED INFORMATION

Functions: `device_map`, `host_adjust_time`, `host_get_time`, `host_set_time`. 
processor_basic_info

**Structure** — Defines the basic information about a processor.

**SYNOPSIS**

```c
[1] struct processor_basic_info
[2] {
[3]   cpu_type_t cpu_type;
[4]   cpu_subtype_t cpu_subtype;
[5]   boolean_t running;
[6]   int slot_num;
[7]   boolean_t is_master;
[8] };
[9] typedef struct processor_basic_info* processor_basic_info_t;
```

**DESCRIPTION**

The `processor_basic_info` structure defines the information available about a processor slot.

**FIELDS**

- `cpu_type`
  Type of CPU
- `cpu_subtype`
  Sub-type of CPU
- `running`
  True if the CPU is running
- `slot_num`
  Slot number of the CPU
- `is_master`
  True if this is the master processor

**RELATED INFORMATION**

Functions: `processor_info`. 
processor_set_basic_info

Structure — Defines the basic information about a processor set.

SYNOPSIS

```
[1] struct processor_set_basic_info
[2] {
[3]     int processor_count;
[4]     int task_count;
[5]     int thread_count;
[6]     int load_average;
[7]     int mach_factor;
[8] }
[9] typedef struct processor_set_basic_info* processor_set_basic_info_t;
```

DESCRIPTION

The processor_set_basic_info structure defines the basic information available about a processor set.

FIELDS

`processor_count`
Number of processors in this set

`task_count`
Number of tasks currently assigned to this processor set

`thread_count`
Number of threads currently assigned to this processor set

`load_average`
Average number of runnable processes divided by number of CPU

`mach_factor`
The processing resources available to a new thread — the number of CPUs divided by (1 + the number of threads)

RELATED INFORMATION

Functions: processor_set_info.

Data Structures: processor_set_sched_info.
processor_set_sched_info

Structure — Defines the scheduling information about a processor set.

SYNOPSIS

```c
[1] struct processor_set_sched_info {
[2]   [3] int policies;
[4]   int max_priority;
[5] };
[6] typedef struct processor_set_sched_info* processor_set_sched_info_t;
```

DESCRIPTION

The `processor_set_sched_info` structure defines the global scheduling information available about a processor set.

FIELDS

- **policies**
  Number of Allowed policies.

- **max_priority**
  Maximum scheduling priority for new threads.

RELATED INFORMATION

Functions: `processor_set_info`.

Data Structures: `processor_set_basic_info`.
sampled_pc

Structure — Defines PC sampling information

SYNOPSIS

```c
[1] struct sampled_pc {
[2]   unsigned int id;
[3]   vm_offset_t pc;
[4]   sampled_pc_flavor_t sampletype;
[5] }
[6] typedef struct sampled_pc sampled_pc_t;
[7] typedef struct sampled_pc* sampled_pc_array_t;
```

DESCRIPTION

The `sampled_pc` structure defines the information provided by the pc sampling routines.

FIELDS

- **id**
  The sampled thread id.

- **pc**
  The sampled pc value.

- **sampletype**
  The sample flavor.

RELATED INFORMATION

Functions: `task_disable_pc_sampling`, `task_enable_pc_sampling`, `task_get_sampled_pc`, `thread_disable_pc_sampling`, `thread_enable_pc_sampling`, `thread_get_sampled_pc`.

Data Structures: `sampled_pc_flavor_t`
security_id_t

Structure — Defines the Security Identifier (SID) structure

SYNOPSIS

```c
[1] struct security_id
[2] {
[3]   mandatory_id_t mid;
[4]   auth_id_t aid;
[5] }
[6] typedef struct security_id security_id_t;
```

DESCRIPTION

The security_id structure defines the label that is associated with subjects and objects in the system.

FIELDS

mid

The mandatory identifier (MID) is a 64-bit field. Its 10 most significant bits define the classifier and can be used by user applications to manage their own objects. The remaining 54 bits are reserved by the Security Server and no structure can be assumed about them.

aid

The authentication identifier (AID) is a 32-bit field.

RELATED INFORMATION

Functions: `SSI_compute_access_vector`, `SSI_context_to_mid`, `SSI_short_context_to_mid`, `SSI_mid_to_context`, `SSI_mid_to_short_context`, `task_change_sid`, any function with `_secure` suffix.
**task_basic_info**

*Structure* — Defines basic information for tasks

**SYNOPSIS**

```c
[1] struct task_basic_info
[2] {
[3]   int suspend_count;
[4]   int base_priority;
[5]   vm_size_t virtual_size;
[6]   vm_size_t resident_size;
[7]   time_value_t user_time;
[8]   time_value_t system_time;
[9] }
[10] typedef struct task_basic_info* task_basic_info_t;
```

**DESCRIPTION**

The *task_basic_info* structure defines the basic information array for tasks. The *task_info* function returns this array for a specified task.

**FIELDS**

- **suspend_count**
  The current suspend count for the task.

- **base_priority**
  The base scheduling priority for the task.

- **virtual_size**
  The number of virtual pages for the task.

- **resident_size**
  The number of resident pages for the task.

- **user_time**
  The total user run time for terminated threads within the task.

- **system_time**
  The total system run time for terminated threads within the task.

**RELATED INFORMATION**

Functions: *task_info*.

Data Structures: *task_thread_times_info*. 

---

Mach 3 Kernel Interfaces
task_basic_secure_info

Structure — Defines basic information including security information for tasks.

SYNOPSIS

```c
[1] struct task_basic_secure_info
[2] {
[3]     int suspend_count;
[4]     int base_priority;
[5]     vm_size_t virtual_size;
[6]     vm_size_t resident_size;
[7]     time_value_t user_time;
[8]     time_value_t system_time;
[9]     security_id_t subj_sid;
[10] }
[11] typedef struct task_basic_secure_info* task_basic_secure_info_t;
```

DESCRIPTION

The task_basic_secure_info structure defines the basic information array, including the task’s subj_sid, for a task. The task_info function returns this array for a specified task.

FIELDS

suspend_count
The current suspend count for the task.

base_priority
The base scheduling priority for the task.

virtual_size
The number of virtual pages for the task.

resident_size
The number of resident pages for the task.

user_time
The total user run time for terminated threads within the task.

system_time
The total system run time for terminated threads within the task.

subj_sid
The task’s associated security identifier.
RELATED INFORMATION

Functions: task_info.

Data Structures: task_thread_times_info.
task_thread_times_info

Structure — Defines thread execution times information for tasks

SYNOPSIS

```c
[1] struct task_thread_times_info
[2] {
[3]     time_value_t user_time;
[4]     time_value_t system_time;
[5] };
[6] typedef struct task_thread_times_info* task_thread_times_info_t;
```

DESCRIPTION

The task_thread_times_info structure defines thread execution time statistics for tasks. The task_info function returns these times for a specified task. The thread_info function returns this information for a specific thread.

FIELDS

- **user_time**
  Total user run time for live threads.

- **system_time**
  Total system run time for live threads.

RELATED INFORMATION

Functions: task_info.

Data Structures: task_basic_info, thread_info.
thread_basic_info

Structure — Defines basic information for threads

SYNOPSIS

```c
[1] struct thread_basic_info
[2] {
[3]     time_value_t user_time;
[4]     time_value_t system_time;
[5]     int cpu_usage;
[6]     int base_priority;
[7]     int cur_priority;
[8]     int run_state;
[9]     int flags;
[10]    int suspend_count;
[12] };
[13] typedef struct thread_basic_info* thread_basic_info_t;
```

DESCRIPTION

The `thread_basic_info` structure defines the basic information array for threads. The `thread_info` function returns this array for a specified thread.

FIELDS

- **user_time**
  
The total user run time for the thread.

- **system_time**
  
The total system run time for the thread.

- **cpu_usage**
  
  Scaled CPU usage percentage for the thread.

- **base_priority**
  
The base scheduling priority for the thread.

- **cur_priority**
  
The current scheduling priority for the thread.

- **run_state**
  
The thread’s run state. Possible values are:

  TH_STATE_RUNNING
    
    The thread is running normally.
TH_STATE_STOPPED
  The thread is stopped.

TH_STATE_WAITING
  The thread is waiting normally.

TH_STATE_UNINTERRUPTIBLE
  The thread is in an un-interruptible wait state.

TH_STATE_HALTED
  The thread is halted at a clean point.

flags
  Swap/idle flags for the thread. Possible values are:

TH_FLAGS_SWAPPED
  The thread is swapped out.

TH_FLAGS_IDLE
  The thread is an idle thread.

suspend_count
  The current suspend count for the thread.

sleep_time
  The number of seconds that the thread has been sleeping.

RELATED INFORMATION
  Functions: thread_info.

  Data Structures: thread_sched_info.
thread_sched_info

Structure — Defines scheduling information for threads

SYNOPSIS

```c
[1] struct thread_sched_info
[2] {
[3]   int policy;
[4]   int data;
[5]   int base_priority;
[6]   int max_priority;
[7]   int cur_priority;
[8]   boolean_t depressed;
[9]   int depress_priority;
[10] }
```

DESCRIPTION

The `thread_sched_info` structure defines the scheduling information array for threads. The `thread_info` function returns this array for a specified thread.

FIELDS

- **policy**
  Scheduling policy in effect

- **data**
  Associated data for the scheduling policy

- **base_priority**
  Base scheduling priority

- **max_priority**
  Maximum scheduling priority

- **cur_priority**
  Current scheduling priority

- **depressed**
  True if scheduling priority is depressed

- **depress_priority**
  Scheduling priority from which depressed
RELATED INFORMATION

Functions: thread_info.

Data Structures: thread_basic_info.
**time_value**

**Structure** — Defines format of system time values

**SYNOPSIS**
```c
[1] struct time_value
[2] {
[3]     long seconds;
[4]     long microseconds;
[5] }
[6] typedef struct time_value time_value_t;
```

**DESCRIPTION**

The *time_value* structure defines the format of the time structure supplied to or returned from the kernel.

**FIELDS**

- **seconds**
  - Seconds since system initialization

- **microseconds**
  - Microseconds in the current second

**RELATED INFORMATION**

Functions: host_adjust_time, host_get_time, host_set_time.
**vm_statistics**

**Structure** — Defines statistics for the kernel’s use of virtual memory

**SYNOPSIS**

```c
[1] struct vm_statistics
[2] {
[3]     long pagesize;
[4]     long free_count;
[5]     long active_count;
[6]     long inactive_count;
[7]     long wire_count;
[8]     long zero_fill_count;
[9]     long reactivations;
[10]    long pageins;
[12]    long faults;
[13]    long cow_faults;
[14]    long lookups;
[15]    long hits;
[16] }
[17] typedef struct vm_statistics* vm_statistics_t;
```

**DESCRIPTION**

The `vm_statistics` structure defines the statistics available on the kernel’s use of virtual memory. The statistics record virtual memory usage since the kernel was booted.

You can also find `pagesize` by using the global variable `vm_page_size`. This variable is set at task initialization and remains constant for the life of the task.

For related information for a specific task, see the `task_basic_info` structure.

**FIELDS**

`pagesize`  
The virtual page size, in bytes.

`free_count`  
The total number of free pages in the system.

`active_count`  
The total number of pages currently in use and pageable.

`inactive_count`  
The number of inactive pages.
**vm_statistics**

wire_count
    The number of pages that are wired in memory and cannot be paged out.

zero_fill_count
    The number of zero-fill pages.

reactivations
    The number of reactivated pages.

pageins
    The number of requests for pages from a pager (such as the i-node pager).

pageouts
    The number of pages that have been paged out.

faults
    The number of times the `vm_fault` routine has been called.

cow_faults
    The number of copy-on-write faults.

lookups
    The number of object cache lookups.

hits
    The number of object cache hits.

**RELATED INFORMATION**

Functions: task_info, vm_statistics.

Data Structures: task_basic_info.
This appendix lists the various kernel return values.

**Error Code Format**

An error code has the following format:

- system code (6 bits). The `err_get_system (err)` macro extracts this field.
- subsystem code (12 bits). The `err_get_sub (err)` macro extracts this field.
- error code (14 bits). The `err_get_code (err)` macro extracts this field.

The various system codes are:

- `err_kern` — kernel
- `err_us` — user space library
- `err_server` — user space servers
- `err_mach_ipc` — Mach-IPC errors
- `err_local` — user defined errors

A typical user error code definition would be:

```c
#define SOMETHING_WRONG err_local | err_sub (13) | 1
```
Error Return Values

MIG Stub Errors

MIG_ARRAY_TOO_LARGE
User specified array not large enough to hold returned array

MIG_BAD_ARGUMENTS
Message receiver found an invalid message size, invalid header fields or invalid descriptors. This could only happen if an invalidly formatted message (i.e., one that did not pass through Mach IPC) were passed to a MIG de-multiplexing routine.

MIG_BAD_ID
Bad message ID. This is only returned by MIG de-multiplexing routines when the message ID in the supplied message is not handled by that routine.

MIG_REPLY_MISMATCH
The message ID in a reply message is not 100 more than the message ID of the request message.

MIG_SERVER_DIED
Message recipient no longer exists, or the recipient destroyed the request message without replying.

MIG_TYPE_ERROR
The wrong number or size of data or rights was received.

Base IPC Status

MACH_MSG_SUCCESS
Normal IPC success. This is the same value as KERN_SUCCESS.

MACH_MSG_IPC_KERNEL
(mask bit) Kernel resource shortage handling an IPC capability.

MACH_MSG_IPC_SPACE
(mask bit) No room in IPC name space for another capability name.
IPC Send Errors

MACH_MSG_VM_KERNEL
(mask bit) Kernel resource shortage handling out-of-line memory.

MACH_MSG_VM_SPACE
(mask bit) No room in VM address space for out-of-line memory.

MACH_MSG_INSUFFICIENT_PERMISSION
(mask bit) A permission check failure prevented the reception of a port right.

IPC Send Errors

The following errors can occur when mach_msg is used with the MACH_SEND_MSG option. This is also the case for all function interfaces.

MACH_SEND_INTERRUPTED
Message send interrupted.

MACH_SEND_INVALID_DATA
Message buffer is unreadable.

MACH_SEND_INVALID_DEST
The destination port name in the message is MACH_PORT_NULL, MACH_PORT_DEAD, names a null or dead right, names a port set or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_SEND_INVALID_HEADER
A field in the message header had a bad value.

MACH_SEND_INVALID_MEMORY
An out-of-line memory region does not exist in the address space or is not readable.

MACH_SEND_INVALID_NOTIFY
The notify port name (MACH_SEND_CANCEL) specified to mach_msg is not a receive right.
MACH_SEND_INVALID_REPLY
The reply port name in the message is MACH_PORT_DEAD, names a null right, names a port set or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_SEND_INVALID_RIGHT
A port name in the message body is MACH_PORT_DEAD, names a null right, names a port set or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_SEND_INVALID_TYPE
Invalid message type specification.

MACH_SEND_MSG_TOO_SMALL
Message buffer doesn’t contain a complete message.

MACH_SEND_NO_BUFFER
No kernel message buffer is available.

MACH_SEND_NO_NOTIFY
Resource shortage; can’t request message-accepted notification.

MACH_SEND_NOTIFY_IN_PROGRESS
Message-accepted notification already pending.

MACH_SEND_TIMED_OUT
Message not sent before time-out expired.

MACH_SEND_WILL_NOTIFY
A message-accepted notification will be generated.

IPC Receive Errors
The following errors can be returned by \texttt{mach_msg} when used with the MACH_RCV_MSG option. They can occur for kernel function interfaces.

MACH_RCV_BODY_ERROR
Error receiving kernel message body. See special bits.
MACH_RCV_HEADER_ERROR
Error receiving message header. See special bits.

MACH_RCV_IN_SET
The receive port name specified to mach_msg is a member of a port set.

MACH_RCV_INTERRUPTED
A software interrupt occurred.

MACH_RCV_INVALID_DATA
The message buffer was not writable.

MACH_RCV_INVALID_NAME
The receive port name specified to mach_msg is MACH_PORT_NULL, MACH_PORT_DEAD, names a null or dead right or is a right whose type (receive, send or send-once) does not match the type specified.

MACH_RCV_INVALID_NOTIFY
The notify port name (MACH_RCV_NOTIFY) specified to mach_msg is not a receive right.

MACH_RCV_PORT_CHANGED
Receive right specified to mach_msg was moved into a set during the receive.

MACH_RCV_PORT_DIED
Receive right (or set) specified to mach_msg was sent away/died during receive.

MACH_RCV_TIMED_OUT
A message was not received within the time-out value.

MACH_RCV_TOO_LARGE
Message buffer is not large enough for the message.

Generic Kernel Errors

KERN_SUCCESS
Successful completion
KERN_INSUFFICIENT_PERMISSION
The requesting task does not have sufficient permission to make the request.

KERN_INVALID_ARGUMENT
The function requested was not applicable to this type of object.

KERN_INVALID_CAPABILITY
The supplied right is dead, null or not of the proper type.

KERN_INVALID_VALUE
A parameter’s value was out of range (or possibly ill-formed). Specific error return values are returned if the parameter’s value is properly formed and in range, but not a usable value at this time.

KERN_RESOURCE_SHORTAGE
A system resource could not be allocated to fulfill this request. This failure may not be permanent.

Port Manipulation Errors

KERN_INVALID_NAME
The port name doesn’t denote a right in the task.

KERN_INVALID_RIGHT
The port name denotes a right, but not an appropriate right.

KERN_NAME_EXISTS
The port name already denotes a right in the task.

KERN_NO_SPACE
The task’s port name space is full.

KERN_NOT_IN_SET
The receive right is not a member of a port set.

KERN_RIGHT_EXISTS
The task already has send or receive rights for the port under another name.
Virtual Memory Manipulation Errors

KERN_UREFS_OVERFLOW
Operation would overflow limit on user-references.

KERN_INVALID_ADDRESS
Specified virtual address is not currently valid.

KERN_MEMORY_ERROR
During a page fault, the memory object indicated that the data could not be returned. This failure may be temporary; future attempts to access this same data may succeed, as defined by the memory object.

KERN_MEMORY_FAILURE
During a page fault, the target address refers to a memory object that has been destroyed. This failure is permanent.

KERN_NO_SPACE
The task’s address space is full (not sufficient free space) or the specified address range is already in use.

KERN_PROTECTION_FAILURE
Specified memory is valid, but does not permit the required forms of access or the protection being requested exceeds that permitted.

Random Kernel Errors

EML_BAD_CNT
Invalid syscall number specified for an emulation vector entry

EML_BAD_TASK
Target of a syscall emulation vector manipulation call is not a task

KERN_ABORTED
The operation was aborted.
Error Return Values

KERN_FAILURE
A catch-all error for implementation dependent failures.

KERN_INVALID_HOST
An argument supplied to assert system privilege was not a host control port.

KERN_INVALID_TASK
Target task isn’t an active task.

Kernel Device Errors

D_SUCCESS
Normal device return. This is the same value as KERN_SUCCESS.

D_ALREADY_OPEN
Exclusive-use device already open

DDEVICE_DOWN
Device has been shut down

D_INVALID_OPERATION
No filter port was specified.

D_INVALID_RECNUM
Invalid record (block) number

D_INVALID_SIZE
Invalid IO size

D_IO_ERROR
Hardware IO error

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.
**Kernel Device Errors**

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

**D_READ_ONLY**
Data cannot be written to this device.

**D_WOULD_BLOCK**
Operation would block, but D_NOWAIT set
Error Return Values
This appendix lists the various permission definitions and their associated values. These permission values are passed from the kernel to the security server to identify which permission is being checked for the given pair. The permission value is also displayed in audit logs, and kernel debugging messages.

**Device Port Permissions**

**DSV_CLOSE_DEVICE**
0x01000011

**DSV_GET_DEVICE_STATUS**
0x01000012

**DSV_MAP_DEVICE**
0x01000013

**DSV_OPEN_DEVICE**
0x01000014

**DSV_READ_DEVICE**
0x01000015
Permission Definitions

DSV_SET DEVICE_FILTER
  0x01000016

DSV_SET_DEVICE_STATUS
  0x01000017

DSV_WRITE_DEVICE
  0x01000018

DSV_PAGER_CTRL
  0x01000019

Host Privileged Port Permissions

HPSV_GET_BOOT_INFO
  0x02000011

HPSV_GET_HOST_PROCESSORS
  0x02000012

HPSV_PSET_CTL_PORT
  0x02000013

HPSV_REBOOT_HOST
  0x02000014

HPSV_SET_DEFAULT_MEMORY_MGR
  0x02000015

HPSV_SET_TIME
  0x02000016

HPSV_WIRE_THREAD
  0x02000017
**Host Port Permissions**

**HPSV_WIRE_VM**
0x02000018

**HSV_CREATE_PSET**
0x03000011

**HSV_FLUSH_PERMISSION**
0x03000012

**HSV_GET_DEFAULT_PSET_NAME**
0x03000013

**HSV_GET_HOST_INFO**
0x03000014

**HSV_GET_HOST_NAME**
0x03000015

**HSV_GET_HOST_VERSION**
0x03000016

**HSV_GET_TIME**
0x03000017

**HSV_PSET_NAMES**
0x03000018

**HSV_GET_AUDIT_PORT**
0x03000019
Permission Definitions

HSV_GET_SECURITY_CLIENT_PORT
0x0300001A

HSV_GET_SECURITY_MASTER_PORT
0x0300001B

HSV_GET_SPECIAL_PORT
0x0300001C

HSV_SET_AUDIT_PORT
0x0300001D

HSV_SET_SECURITY_CLIENT_PORT
0x0300001E

HSV_SET_SECURITY_MASTER_PORT
0x0300001F

HSV_SET_SPECIAL_PORT
0x03000020

HSV_GET_CRYPTO_PORT
0x03000021

HSV_GET_HOST_CONTROL_PORT
0x03000022

HSV_GET_NEGOTIATION_PORT
0x03000023

HSV_SET_CRYPTO_PORT
0x03000024
HSV_SET_NEGOTIATION_PORT
  0x03000025

HSV_GET_AUTHENTICATION_PORT
  0x03000026

HSV_SET_AUTHENTICATION_PORT
  0x03000027

Kernel Reply Port Permissions

KRPSV_PROVIDE_PERMISSION
  0x0B000011

Memory Object Permissions

MOSV_HAVE_EXECUTE
  0x04000011

MOSV_HAVE_READ
  0x04000012

MOSV_HAVE_WRITE
  0x04000013

MOSV_PAGE_VM_REGION
  0x04000015

Memory Control Port Permissions

MCSV_CHANGE_PAGE_LOCKS
  0x05000011
MCSV_DESTROY_OBJECT
0x05000012

MCSV_GET_ATTRIBUTE
0x05000013

MCSV_INVoke_LOCK_REQUEST
0x05000014

MCSV_MAKE_PAGE_PRECIOUS
0x05000015

MCSV_PROVIDE_DATA
0x05000016

MCSV_REMOVE_PAGE
0x05000017

MCSV_REVOKE_IBAC
0x05000018

MCSV_SAVE_PAGE
0x05000019

MCSV_SET_ATTRIBUTES
0x0500001a

MCSV_SET_IBAC_PORT
0x0500001b

MCSV_SUPPLY_IBAC
0x0500001c
Processor Port Permissions

PSV_ASSIGN_PROCESSOR_TO_SET
0x06000011

PSV_GET_PROCESSOR_ASSIGNMENT
0x06000012

PSV_GET_PROCESSOR_INFO
0x06000013

PSV_MAY_CONTROL_PROCESSOR
0x06000014

Processor Set Permissions

PSSV_ASSIGN_PROCESSOR
0x07000011

PSSV_ASSIGN_TASK
0x07000012

PSSV_ASSIGN_THREAD
0x07000013

PSSV_CHG_PSET_MAX_PRI
0x07000014

PSSV_DEFINE_NEW_SCHEDULING_POLICY
0x07000015
Permission Definitions

PSSV_DESTROY_PSET
  0x07000016

PSSV_GET_PSET_INFO
  0x07000017

PSSV_INVALIDATE_SCHEDULING_POLICY
  0x07000018

PSSV_OBSERVE_PSET PROCESSES
  0x07000019

Task Port Permissions

TSV_ACCESS_MACHINE_ATTRIBUTE
  0x08000011

TSV_ADD_NAME
  0x08000012

TSV_ADD_THREAD
  0x08000013

TSV_ADD_THREAD SERVICE
  0x08000014

TSV_ALLOCATE_VM_REGION
  0x08000015

TSV_ALTER_PNS_INFO
  0x08000016
Task Port Permissions

TSV_ASSIGN_TASK_TO_PSET
  0x08000017

TSV_CHG_VM_REGION_PROT
  0x08000018

TSV_CHG_TASK_PRIORITY
  0x08000019

TSV_COPY_VM
  0x0800001a

TSV_CREATE_TASK
  0x0800001b

TSV_CREATE_TASK_SECURE
  0x0800001c

TSV_DEALLOCATE_VM_REGION
  0x0800001d

TSV_EXTRACT_RIGHT
  0x0800001e

TSV_GET_EMULATION
  0x0800001f

TSV_GET_TASK_ASSIGNMENT
  0x08000020
Permission Definitions

TSV_GET_TASK_BOOT_PORT
0x08000021

TSV_GET_TASK_EXCEPTION_PORT
0x08000022

TSV_GET_TASK_INFO
0x08000023

TSV_GET_TASK_KERNEL_PORT
0x08000024

TSV_GET_TASK_THREADS
0x08000025

TSV_GET_VM_REGION_INFO
0x08000026

TSV_GET_VM_STATISTICS
0x08000027

TSV_LOOKUP_PORTS
0x08000028

TSV_MANIPULATE_PORT_SET
0x08000029

TSV_OBSERVE_PNS_INFO
0x0800002a

TSV_PORT_RENAME
0x0800002b
Task Port Permissions

TSV_READ_VM_REGION
    0x0800002c

TSV_REGISTER_NOTIFICATION
    0x0800002d

TSV_REGISTER_PORTS
    0x0800002e

TSV_REMOVE_NAME
    0x0800002f

TSV_RESUME_TASK
    0x08000030

TSV_SAMPLE_TASK
    0x08000031

TSV_SET_EMULATION
    0x08000032

TSV_SET_VM_REGION_INHERIT
    0x08000033

TSV_SET_RAS
    0x08000034

TSV_SET_TASK_BOOT_PORT
    0x08000035
TSV_SET_TASK_EXCEPTION_PORT
    0x08000036

TSV_SET_TASK_KERNEL_PORT
    0x08000037

TSV_SUSPEND_TASK
    0x08000038

TSV_TERMINATE_TASK
    0x08000039

TSV_WIRE_VM_FOR_TASK
    0x0800003a

TSV_WRITE_VM_REGION
    0x0800003b

TSV_CROSS_CONTEXT_CREATE
    0x0800003c

TSV_CROSS_CONTEXT_INHERIT
    0x0800003d

TSV_CHG_SID
    0x0800003e

TSV_MAKE_SID
    0x0800003f

TSV_TRANSITION_SID
    0x08000040
Thread Port Permissions

THSV_ABORT_THREAD
0x09000011

THSV_ABORT_THREAD_DEPRESS
0x09000012

THSV_ASSIGN_THREAD_TO_PSET
0x09000013

THSV_CAN_SWTCH
0x09000014

THSV_CAN_SWTCH_PRI
0x09000015

THSV_DEPRESS_PRI
0x09000016

THSV_GET_THREAD_ASSIGNMENT
0x09000017

THSV_GET_THREAD_EXCEPTION_PORT
0x09000018

THSV_GET_THREAD_INFO
0x09000019

THSV_GET_THREAD_KERNEL_PORT
0x0900001a
THSV_GET_THREAD_STATE
          0x0900001b

THSV_INITIATE_SECURE
          0x0900001c

THSV_RAISE_EXCEPTION
          0x0900001d

THSV_RESUME_THREAD
          0x0900001e

THSV_SAMPLE_THREAD
          0x0900001f

THSV_SET_MAX_THREAD_PRIORITY
          0x09000020

THSV_SET_THREAD_EXCEPTION_PORT
          0x09000021

THSV_SET_THREAD_KERNEL_PORT
          0x09000022

THSV_SET_THREAD_POLICY
          0x09000023

THSV_SET_THREAD_PRIORITY
          0x09000024

THSV_SET_THREAD_STATE
          0x09000025
Mach 3 Kernel Interfaces

IPC Permissions

THSV_SUSPEND_THREAD
0x09000026

THSV_SWITCH_THREAD
0x09000027

THSV_TERMINATE_THREAD
0x09000028

THSV_WAIT_EVC
0x09000029

THSV_WIRE_THREAD_INTO_MEMORY
0x0900002a

IPC Permissions

AV_CAN.Receive
0x0a000001

AV.Receive
0x0a000001

AV_CAN.Send
0x0a000002

AV.Send
0x0a000002

AV_HOLD.Receive
0x0a000003
Permission Definitions

AV_HOLD_SEND
0x0a000004

AV_HOLD_SEND_ONCE
0x0a000005

AV_INTERPOSE
0x0a000006

AV_SPECIFY
0x0a000007

AV_TRANSFER_OOL
0x0a000008

AV_TRANSFER_RECEIVE
0x0a000009

AV_TRANSFER_SEND
0x0a00000a

AV_TRANSFER_SEND_ONCE
0x0a00000b

AV_TRANSFER_RIGHTS
0x0a00000c

MOSV_MAP_VM_REGION
0x0a00000d

AV_SET_REPLY
0x0a00000e
IPC Permissions
Permission Definitions
## Object Index

### A
- **abstract memory object**
  - `vm_map` .................................................. 93
  - `memory_object_copy` ................................. 114
  - `memory_object_data_request` .................. 121
  - `memory_object_data_return` .................... 123
  - `memory_object_data_unlock` .................... 130
  - `memory_object_data_write` ....................... 132
  - `memory_object_init` ............................... 138
  - `memory_object_terminate` ....................... 153
  - `device_map` ............................................ 277
  - `default_pager_object_create` .................. 350
  - `memory_object_create` ............................ 352
  - `memory_object_data_initialize` ............... 355

### B
- **bootstrap**
  - `task_create/task_create_secure` ................. 195
  - `task_get_special_port` .......................... 200
  - `task_set_special_port` ......................... 211
  - `norma_task_clone` ................................. 367
  - `norma_task_create` ............................... 369

### D
- **default pager**
  - `default_pager_info` ............................... 348
  - `default_pager_object_create` .................. 350
  - `memory_object_create` ............................ 352
  - `vm_set_default_memory_manager` ............... 357
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>norma_get_special_port</td>
<td>360</td>
</tr>
<tr>
<td>norma_set_special_port</td>
<td>364</td>
</tr>
<tr>
<td>device_close</td>
<td>274</td>
</tr>
<tr>
<td>device_get_status</td>
<td>275</td>
</tr>
<tr>
<td>device_map</td>
<td>277</td>
</tr>
<tr>
<td>device_open</td>
<td>279</td>
</tr>
<tr>
<td>device_read</td>
<td>282</td>
</tr>
<tr>
<td>device_read_inband</td>
<td>285</td>
</tr>
<tr>
<td>device_set_filter</td>
<td>288</td>
</tr>
<tr>
<td>device_set_status</td>
<td>292</td>
</tr>
<tr>
<td>device_write</td>
<td>294</td>
</tr>
<tr>
<td>device_write_inband</td>
<td>297</td>
</tr>
<tr>
<td>i386_io_port_add</td>
<td>378</td>
</tr>
<tr>
<td>i386_io_port_list</td>
<td>379</td>
</tr>
<tr>
<td>i386_io_port_remove</td>
<td>380</td>
</tr>
<tr>
<td>device_master device_open</td>
<td>279</td>
</tr>
<tr>
<td>norma_get_special_port</td>
<td>360</td>
</tr>
<tr>
<td>norma_set_special_port</td>
<td>364</td>
</tr>
<tr>
<td>exception catch_exception_raise</td>
<td>156</td>
</tr>
<tr>
<td>thread_get_special_port</td>
<td>169</td>
</tr>
<tr>
<td>thread_set_special_port</td>
<td>178</td>
</tr>
<tr>
<td>task_create/task_create_secure</td>
<td>195</td>
</tr>
<tr>
<td>task_get_special_port</td>
<td>200</td>
</tr>
<tr>
<td>task_set_special_port</td>
<td>211</td>
</tr>
<tr>
<td>norma_task_clone</td>
<td>367</td>
</tr>
<tr>
<td>norma_task_create</td>
<td>369</td>
</tr>
<tr>
<td>filter device_set_filter</td>
<td>288</td>
</tr>
<tr>
<td>host control vm_wire</td>
<td>105</td>
</tr>
<tr>
<td>thread_wire</td>
<td>186</td>
</tr>
<tr>
<td>host_adjust_time</td>
<td>218</td>
</tr>
<tr>
<td>host_get_boot_info</td>
<td>219</td>
</tr>
<tr>
<td>host_get_special_port</td>
<td>220</td>
</tr>
<tr>
<td>host_reboot</td>
<td>226</td>
</tr>
<tr>
<td>host_set_special_port</td>
<td>227</td>
</tr>
<tr>
<td>host_set_time</td>
<td>229</td>
</tr>
<tr>
<td>host_processor_set_priv</td>
<td>232</td>
</tr>
<tr>
<td>host_processors</td>
<td>235</td>
</tr>
<tr>
<td>vm_set_default_memory_manager</td>
<td>357</td>
</tr>
<tr>
<td>norma_get_special_port</td>
<td>360</td>
</tr>
<tr>
<td>norma_set_special_port</td>
<td>364</td>
</tr>
<tr>
<td><strong>host name</strong></td>
<td><strong>memory cache control</strong></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>host_get_time</td>
<td>memory_object_change_attributes</td>
</tr>
</tbody>
</table>
Object Index

mach_msg_send ........................................... 27
do_mach_notify_dead_name ............................... 30
do_mach_notify_msg_accepted ......................... 32
do_mach_notify_no_senders ............................. 34
do_mach_notify_port_deleted ........................... 36
do_mach_notify_port_destroyed ....................... 38
do_mach_notify_send_once ............................... 40
mach_port_request_notification ....................... 68

P
processor
  host_processors........................................ 235
  processor_assign...................................... 236
  processor_control..................................... 238
  processor_exit........................................ 240
  processor_get_assignment............................ 241
  processor_info....................................... 242
  processor_start...................................... 257

processor_set control
  host_processor_set_priv ............................. 232
  processor_assign...................................... 236
  processor_set_create................................ 244
  processor_set_destroy................................ 247
  processor_set_info................................... 248
  processor_set_max_priority........................ 250
  processor_set_policy_disable....................... 252
  processor_set_policy_enable......................... 254
  processor_set_tasks.................................. 255
  processor_set_threads................................ 256
  task_assign.......................................... 258
  thread_assign....................................... 265
  thread_max_priority................................. 268

processor_set name
  host_processor_set_priv ............................. 232
  host_processor_sets................................ 233
  processor_get_assignment............................ 241
  processor_set_create................................ 244
  processor_set_default................................ 246
  processor_set_info................................... 248
  task_get_assignment.................................. 262
  thread_get_assignment................................ 267

R
random
  mach_msg/mach_msg_secure............................ 8
  mach_msg_receive.................................... 26
  mach_msg_send....................................... 27
do_mach_notify_port_destroyed...................... 38
  mach_port_extract_right............................ 50
  mach_port_insert_right.............................. 58
| registered                              | mach_ports_lookup                          | 190 |
|                                       | mach_ports_register                        | 191 |
|                                       | task_create/task_create_secure             | 195 |
|                                       | norma_task_clone                           | 367 |
|                                       | norma_task_create                          | 369 |
| reply                                  | mach_msg/mach_msg_secure                   | 8   |
|                                       | mach_msg_receive                           | 26  |
|                                       | mach_msg_send                              | 27  |
|                                       | mach_reply_port                            | 79  |
|                                       | memory_object_change_attributes            | 110 |
|                                       | memory_object_change_completed             | 112 |
|                                       | memory_object_data_supply                  | 125 |
|                                       | memory_object_lock_completed               | 141 |
|                                       | memory_object_lock_request                 | 143 |
|                                       | memory_object_supply_completed             | 151 |
|                                       | device_read                                | 282 |
|                                       | device_read_inband                         | 285 |
|                                       | device_write                               | 294 |
|                                       | device_write_inband                        | 297 |
| S sample                               | receive_samples                            | 160 |
|                                       | thread_sample                              | 176 |
|                                       | task_create/task_create_secure             | 195 |
|                                       | task_sample                                | 205 |
|                                       | norma_task_clone                           | 367 |
|                                       | norma_task_create                          | 369 |
| T task                                 | mach_port_allocate/mach_port_allocate_secure | 41  |
|                                       | mach_port_allocate_name/                   |     |
|                                       | mach_port_allocate_name_secure             | 44  |
|                                       | mach_port_deallocate                       | 47  |
|                                       | mach_port_destroy                         | 48  |
|                                       | mach_port_extract_right                    | 50  |
|                                       | mach_port_get_receive_status               | 52  |
|                                       | mach_port_get.refs                        | 54  |
|                                       | mach_port_get_set_status                   | 56  |
|                                       | mach_port_insert_right                     | 58  |
|                                       | mach_port_mod_refs                         | 60  |
|                                       | mach_port_move_member                      | 62  |
|                                       | mach_port_names                            | 64  |
|                                       | mach_port_rename                          | 66  |
|                                       | mach_port_request_notification             | 68  |
|                                       | mach_port_set_mscount                      | 71  |
|                                       | mach_port_set qlimit                       | 73  |
### Object Index

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mach_port_set_seqno</td>
<td>75</td>
</tr>
<tr>
<td>mach_port_type/mach_port_type_secure</td>
<td>77</td>
</tr>
<tr>
<td>vm_allocate/vm_allocate_secure</td>
<td>82</td>
</tr>
<tr>
<td>vm_copy</td>
<td>85</td>
</tr>
<tr>
<td>vm_deallocate</td>
<td>87</td>
</tr>
<tr>
<td>vm_inherit</td>
<td>89</td>
</tr>
<tr>
<td>vm_machine_attribute</td>
<td>91</td>
</tr>
<tr>
<td>vm_map</td>
<td>93</td>
</tr>
<tr>
<td>vm_protect</td>
<td>97</td>
</tr>
<tr>
<td>vm_read</td>
<td>99</td>
</tr>
<tr>
<td>vm_region/vm_region_secure</td>
<td>101</td>
</tr>
<tr>
<td>vm_statistics</td>
<td>104</td>
</tr>
<tr>
<td>vm_wire</td>
<td>105</td>
</tr>
<tr>
<td>vm_write</td>
<td>107</td>
</tr>
<tr>
<td>catch_exception_raise</td>
<td>156</td>
</tr>
<tr>
<td>thread_create/thread_create_secure</td>
<td>166</td>
</tr>
<tr>
<td>mach_ports_lookup</td>
<td>190</td>
</tr>
<tr>
<td>mach_ports_register</td>
<td>191</td>
</tr>
<tr>
<td>task_create/task_create_secure</td>
<td>195</td>
</tr>
<tr>
<td>task_get_emulation_vector</td>
<td>198</td>
</tr>
<tr>
<td>task_get_special_port</td>
<td>200</td>
</tr>
<tr>
<td>task_info</td>
<td>202</td>
</tr>
<tr>
<td>task_resume</td>
<td>204</td>
</tr>
<tr>
<td>task_sample</td>
<td>205</td>
</tr>
<tr>
<td>task_set_emulation</td>
<td>207</td>
</tr>
<tr>
<td>task_set_emulation_vector</td>
<td>209</td>
</tr>
<tr>
<td>task_set_special_port</td>
<td>211</td>
</tr>
<tr>
<td>task_suspend</td>
<td>213</td>
</tr>
<tr>
<td>taskTerminate</td>
<td>214</td>
</tr>
<tr>
<td>task_threads</td>
<td>215</td>
</tr>
<tr>
<td>processor_set_tasks</td>
<td>255</td>
</tr>
<tr>
<td>task_assign</td>
<td>258</td>
</tr>
<tr>
<td>task_assign_default</td>
<td>260</td>
</tr>
<tr>
<td>task_get_assignment</td>
<td>262</td>
</tr>
<tr>
<td>task_priority</td>
<td>263</td>
</tr>
<tr>
<td>norma_port_location_hint</td>
<td>363</td>
</tr>
<tr>
<td>norma_task_clone</td>
<td>367</td>
</tr>
<tr>
<td>norma_task_create</td>
<td>369</td>
</tr>
<tr>
<td>task_set_child_node</td>
<td>371</td>
</tr>
<tr>
<td>mach_task_self</td>
<td>193</td>
</tr>
<tr>
<td>task_create/task_create_secure</td>
<td>195</td>
</tr>
<tr>
<td>task_get_special_port</td>
<td>200</td>
</tr>
<tr>
<td>task_set_special_port</td>
<td>211</td>
</tr>
<tr>
<td>norma_task_clone</td>
<td>367</td>
</tr>
<tr>
<td>norma_task_create</td>
<td>369</td>
</tr>
<tr>
<td><strong>task special</strong></td>
<td>task_get_special_port</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td></td>
<td>task_set_special_port</td>
</tr>
<tr>
<td><strong>thread</strong></td>
<td>catch_exception_raise</td>
</tr>
<tr>
<td></td>
<td>thread_abort</td>
</tr>
<tr>
<td></td>
<td>thread_create/thread_create_secure</td>
</tr>
<tr>
<td></td>
<td>thread_depress_abort</td>
</tr>
<tr>
<td></td>
<td>thread_get_special_port</td>
</tr>
<tr>
<td></td>
<td>thread_get_state</td>
</tr>
<tr>
<td></td>
<td>thread_info</td>
</tr>
<tr>
<td></td>
<td>thread_resume/thread_resume_secure</td>
</tr>
<tr>
<td></td>
<td>thread_sample</td>
</tr>
<tr>
<td></td>
<td>thread_set_special_port</td>
</tr>
<tr>
<td></td>
<td>thread_set_state/thread_set_state_secure</td>
</tr>
<tr>
<td></td>
<td>thread_suspend</td>
</tr>
<tr>
<td></td>
<td>thread_switch</td>
</tr>
<tr>
<td></td>
<td>thread_terminate</td>
</tr>
<tr>
<td></td>
<td>thread_wire</td>
</tr>
<tr>
<td></td>
<td>task_threads</td>
</tr>
<tr>
<td></td>
<td>processor_set_threads</td>
</tr>
<tr>
<td></td>
<td>thread_assign</td>
</tr>
<tr>
<td></td>
<td>thread_assign_default</td>
</tr>
<tr>
<td></td>
<td>thread_get_assignment</td>
</tr>
<tr>
<td></td>
<td>thread_max_priority</td>
</tr>
<tr>
<td></td>
<td>thread_policy</td>
</tr>
<tr>
<td></td>
<td>thread_priority</td>
</tr>
<tr>
<td></td>
<td>i386_get_ldt</td>
</tr>
<tr>
<td></td>
<td>i386_io_port_add</td>
</tr>
<tr>
<td></td>
<td>i386_io_port_list</td>
</tr>
<tr>
<td></td>
<td>i386_io_port_remove</td>
</tr>
<tr>
<td></td>
<td>i386_set_ldt</td>
</tr>
<tr>
<td><strong>thread self</strong></td>
<td>mach_thread_self</td>
</tr>
<tr>
<td></td>
<td>thread_get_special_port</td>
</tr>
<tr>
<td></td>
<td>thread_set_special_port</td>
</tr>
<tr>
<td><strong>thread special</strong></td>
<td>thread_get_special_port</td>
</tr>
<tr>
<td></td>
<td>thread_set_special_port</td>
</tr>
<tr>
<td>Object Index</td>
<td></td>
</tr>
</tbody>
</table>
## APPENDIX I

### Interface and Structure Index

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base IPC Status</td>
<td>436</td>
</tr>
<tr>
<td>Data Structures</td>
<td>383</td>
</tr>
<tr>
<td>Default Memory Management Interface</td>
<td>347</td>
</tr>
<tr>
<td>Device Port Permissions</td>
<td>445</td>
</tr>
<tr>
<td>Error Code Format</td>
<td>435</td>
</tr>
<tr>
<td>Error Return Values</td>
<td>4</td>
</tr>
<tr>
<td>Error Return Values</td>
<td>435</td>
</tr>
<tr>
<td>External Memory Management Interface</td>
<td>109</td>
</tr>
<tr>
<td>Generic Kernel Errors</td>
<td>439</td>
</tr>
<tr>
<td>Host Interface</td>
<td>217</td>
</tr>
<tr>
<td>Host Port Permissions</td>
<td>447</td>
</tr>
<tr>
<td>Host Privileged Port Permissions</td>
<td>446</td>
</tr>
<tr>
<td>IPC Interface</td>
<td>7</td>
</tr>
<tr>
<td>IPC Permissions</td>
<td>459</td>
</tr>
<tr>
<td>IPC Receive Errors</td>
<td>438</td>
</tr>
<tr>
<td>IPC Send Errors</td>
<td>437</td>
</tr>
<tr>
<td>Intel 386 Support</td>
<td>373</td>
</tr>
<tr>
<td>Interface Descriptions</td>
<td>1</td>
</tr>
<tr>
<td>Interface Types</td>
<td>2</td>
</tr>
<tr>
<td>Interface and Structure Index</td>
<td>471</td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Kernel Device Errors</td>
<td>442</td>
</tr>
<tr>
<td>Kernel Device Interface</td>
<td>273</td>
</tr>
<tr>
<td>Kernel Reply Port Permissions</td>
<td>449</td>
</tr>
<tr>
<td>MIG Server Routines</td>
<td>329</td>
</tr>
<tr>
<td>MIG Stub Errors</td>
<td>436</td>
</tr>
<tr>
<td>Memory Control Port Permissions</td>
<td>449</td>
</tr>
<tr>
<td>Memory Object Permissions</td>
<td>449</td>
</tr>
<tr>
<td>Multicomputer Support</td>
<td>359</td>
</tr>
<tr>
<td>Object Index</td>
<td>463</td>
</tr>
<tr>
<td>Parameter Types</td>
<td>3</td>
</tr>
<tr>
<td>Permission Definitions</td>
<td>445</td>
</tr>
<tr>
<td>Port Manipulation Errors</td>
<td>440</td>
</tr>
<tr>
<td>Port Manipulation Interface</td>
<td>29</td>
</tr>
<tr>
<td>Processor Management and Scheduling Interface</td>
<td>231</td>
</tr>
<tr>
<td>Processor Port Permissions</td>
<td>451</td>
</tr>
<tr>
<td>Processor Set Permissions</td>
<td>451</td>
</tr>
<tr>
<td>Random Kernel Errors</td>
<td>441</td>
</tr>
<tr>
<td>SSL_compute_access_vector</td>
<td>309</td>
</tr>
<tr>
<td>SSL_context_to_mid</td>
<td>312</td>
</tr>
<tr>
<td>SSL_load_security_policy</td>
<td>314</td>
</tr>
<tr>
<td>SSL_mid_to_context</td>
<td>320</td>
</tr>
<tr>
<td>SSL_mid_to_short_context</td>
<td>322</td>
</tr>
<tr>
<td>SSL_record_name_server</td>
<td>315</td>
</tr>
<tr>
<td>SSL_register_caching_server</td>
<td>316</td>
</tr>
<tr>
<td>SSL_short_context_to_mid</td>
<td>318</td>
</tr>
<tr>
<td>SSL_transfer_security_server_ports</td>
<td>324</td>
</tr>
<tr>
<td>SSL_transition_domain</td>
<td>326</td>
</tr>
<tr>
<td>Security Controls</td>
<td>5</td>
</tr>
<tr>
<td>Security Server Interface</td>
<td>303</td>
</tr>
<tr>
<td>Special Forms</td>
<td>3</td>
</tr>
<tr>
<td>Task Interface</td>
<td>189</td>
</tr>
<tr>
<td>Task Port Permissions</td>
<td>452</td>
</tr>
<tr>
<td>Interface and Structure Index</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td></td>
</tr>
<tr>
<td>Thread Interface ...........</td>
<td>155</td>
</tr>
<tr>
<td>Thread Port Permissions ....</td>
<td>457</td>
</tr>
<tr>
<td>Virtual Memory Interface ...</td>
<td>81</td>
</tr>
<tr>
<td>Virtual Memory Manipulation Errors</td>
<td>441</td>
</tr>
<tr>
<td>avc_cache_control, avc_cache_control_trap</td>
<td>304</td>
</tr>
<tr>
<td>catch_exception_raise .....</td>
<td>156</td>
</tr>
<tr>
<td>default_pager_info ..........</td>
<td>348</td>
</tr>
<tr>
<td>default_pager_object_create</td>
<td>350</td>
</tr>
<tr>
<td>device_close ...............</td>
<td>274</td>
</tr>
<tr>
<td>device_get_status ..........</td>
<td>275</td>
</tr>
<tr>
<td>device_map ..................</td>
<td>277</td>
</tr>
<tr>
<td>device_open .................</td>
<td>279</td>
</tr>
<tr>
<td>device_open_request .......</td>
<td>279</td>
</tr>
<tr>
<td>device_read .................</td>
<td>282</td>
</tr>
<tr>
<td>device_read_inband ..........</td>
<td>285</td>
</tr>
<tr>
<td>device_read_request .......</td>
<td>282</td>
</tr>
<tr>
<td>device_read_request_inband</td>
<td>285</td>
</tr>
<tr>
<td>device_reply_server .......</td>
<td>330</td>
</tr>
<tr>
<td>device_set_filter ..........</td>
<td>288</td>
</tr>
<tr>
<td>device_set_status ..........</td>
<td>292</td>
</tr>
<tr>
<td>device_write ...............</td>
<td>294</td>
</tr>
<tr>
<td>device_write_inband .......</td>
<td>297</td>
</tr>
<tr>
<td>device_write_request .......</td>
<td>294</td>
</tr>
<tr>
<td>device_write_request_inband</td>
<td>297</td>
</tr>
<tr>
<td>do_mach_notify_dead_name ...</td>
<td>30</td>
</tr>
<tr>
<td>do_mach_notify_msg_accepted</td>
<td>32</td>
</tr>
<tr>
<td>do_mach_notify_no_senders</td>
<td>34</td>
</tr>
<tr>
<td>do_mach_notify_port_destroyed</td>
<td>38</td>
</tr>
<tr>
<td>do_mach_notify_send_once ...</td>
<td>40</td>
</tr>
<tr>
<td>do_seqnos_mach_notify_dead_name</td>
<td>30</td>
</tr>
<tr>
<td>do_seqnos_mach_notify_msg_accepted</td>
<td>32</td>
</tr>
<tr>
<td>do_seqnos_mach_notify_no_senders</td>
<td>34</td>
</tr>
<tr>
<td>do_seqnos_mach_notify_port_deleted</td>
<td>36</td>
</tr>
<tr>
<td>ds_device_open_reply .......</td>
<td>279</td>
</tr>
<tr>
<td>ds_device_read_reply .......</td>
<td>282</td>
</tr>
<tr>
<td>ds_device_read_reply_inband</td>
<td>285</td>
</tr>
<tr>
<td>ds_device_write_reply ......</td>
<td>294</td>
</tr>
<tr>
<td>ds_device_write_reply_inband</td>
<td>297</td>
</tr>
<tr>
<td>evc_wait .............</td>
<td>300</td>
</tr>
<tr>
<td>exc_server ...............</td>
<td>332</td>
</tr>
<tr>
<td>extract_aid ..........</td>
<td>306</td>
</tr>
<tr>
<td>extract_mid ..............</td>
<td>307</td>
</tr>
<tr>
<td>host_adjust_time ..........</td>
<td>218</td>
</tr>
<tr>
<td>host_basic_info ..........</td>
<td>384</td>
</tr>
<tr>
<td>host_get_boot_info .......</td>
<td>219</td>
</tr>
<tr>
<td>host_get_special_port .....</td>
<td>220</td>
</tr>
<tr>
<td>host_get_time ..........</td>
<td>222</td>
</tr>
<tr>
<td>host_info .............</td>
<td>223</td>
</tr>
<tr>
<td>host_kernel_version .......</td>
<td>225</td>
</tr>
<tr>
<td>host_load_info ..........</td>
<td>385</td>
</tr>
<tr>
<td>host_process_set_priv ....</td>
<td>232</td>
</tr>
<tr>
<td>host_processors ..........</td>
<td>233</td>
</tr>
<tr>
<td>host_reboot ..........</td>
<td>226</td>
</tr>
<tr>
<td>host_sched_info ..........</td>
<td>386</td>
</tr>
<tr>
<td>host_set_special_port ....</td>
<td>227</td>
</tr>
<tr>
<td>host_set_time ..........</td>
<td>229</td>
</tr>
<tr>
<td>i386_get_1dt .............</td>
<td>376</td>
</tr>
<tr>
<td>i386_io_port_add ..........</td>
<td>378</td>
</tr>
<tr>
<td>i386_io_port_list ........</td>
<td>379</td>
</tr>
<tr>
<td>i386_io_port_remove .....</td>
<td>380</td>
</tr>
<tr>
<td>i386_set_1dt ............</td>
<td>381</td>
</tr>
<tr>
<td>mach_access_vector .......</td>
<td>387</td>
</tr>
<tr>
<td>mach_device_services .......</td>
<td>390</td>
</tr>
<tr>
<td>mach_generic_services .......</td>
<td>391</td>
</tr>
<tr>
<td>mach_host_priv_services ....</td>
<td>393</td>
</tr>
<tr>
<td>mach_host_set_self .......</td>
<td>230</td>
</tr>
<tr>
<td>mach_host_services .......</td>
<td>394</td>
</tr>
<tr>
<td>mach_kernel_reply_port_services</td>
<td>392</td>
</tr>
<tr>
<td>mach_mem_ctrl_services ....</td>
<td>397</td>
</tr>
<tr>
<td>mach_mem_obj_services .......</td>
<td>396</td>
</tr>
<tr>
<td>mach_msg/mach_msg_secure</td>
<td>8</td>
</tr>
<tr>
<td>mach_msg_header .........</td>
<td>399</td>
</tr>
<tr>
<td>mach_msg_receive .......</td>
<td>26</td>
</tr>
<tr>
<td>mach_msg_send ..........</td>
<td>27</td>
</tr>
<tr>
<td>mach_msg_type ..........</td>
<td>402</td>
</tr>
<tr>
<td>mach_msg_type_long ......</td>
<td>405</td>
</tr>
<tr>
<td>mach_port_allocate .......</td>
<td>41</td>
</tr>
<tr>
<td>mach_port_allocate_secure</td>
<td>41</td>
</tr>
<tr>
<td>mach_port_allocate_name/</td>
<td>44</td>
</tr>
<tr>
<td>mach_port_allocate_name_secure</td>
<td>44</td>
</tr>
<tr>
<td>mach_port_deallocate ....</td>
<td>47</td>
</tr>
<tr>
<td>mach_port_destroy ......</td>
<td>48</td>
</tr>
<tr>
<td>mach_port_extract_right</td>
<td>50</td>
</tr>
<tr>
<td>mach_port_get_receive_status</td>
<td>52</td>
</tr>
<tr>
<td>mach_port_get_refs ......</td>
<td>54</td>
</tr>
<tr>
<td>mach_port_get_set_status</td>
<td>56</td>
</tr>
<tr>
<td>mach_port_insert_right</td>
<td>58</td>
</tr>
<tr>
<td>mach_port_mod_refs ......</td>
<td>60</td>
</tr>
<tr>
<td>mach_port_move_member</td>
<td>62</td>
</tr>
<tr>
<td>mach_port_names ........</td>
<td>64</td>
</tr>
<tr>
<td>mach_port_rename .......</td>
<td>66</td>
</tr>
<tr>
<td>mach_port_request_notification</td>
<td>68</td>
</tr>
<tr>
<td>mach_port_set_msgcount</td>
<td>71</td>
</tr>
<tr>
<td>mach_port_set_qlimit</td>
<td>73</td>
</tr>
<tr>
<td>Function/Method</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
</tr>
<tr>
<td>mach_port_set_seqno</td>
<td>75</td>
</tr>
<tr>
<td>mach_port_status</td>
<td>407</td>
</tr>
<tr>
<td>mach_port_type</td>
<td></td>
</tr>
<tr>
<td>mach_port_type_secure</td>
<td>77</td>
</tr>
<tr>
<td>mach_ports_lookup</td>
<td>190</td>
</tr>
<tr>
<td>mach_ports_register</td>
<td>191</td>
</tr>
<tr>
<td>mach_proc_services</td>
<td>409</td>
</tr>
<tr>
<td>mach_proc_set_services</td>
<td>410</td>
</tr>
<tr>
<td>mach_reply_port</td>
<td>79</td>
</tr>
<tr>
<td>mach_services</td>
<td>411</td>
</tr>
<tr>
<td>mach_task_self</td>
<td>193</td>
</tr>
<tr>
<td>mach_task_services</td>
<td>413</td>
</tr>
<tr>
<td>mach_thread_self</td>
<td>159</td>
</tr>
<tr>
<td>mach_thread_services</td>
<td>415</td>
</tr>
<tr>
<td>make_sid</td>
<td>308</td>
</tr>
<tr>
<td>mapped_time_value</td>
<td>417</td>
</tr>
<tr>
<td>memory_object_change_attributes</td>
<td>110</td>
</tr>
<tr>
<td>memory_object_change_completed</td>
<td>112</td>
</tr>
<tr>
<td>memory_object_copy</td>
<td>114</td>
</tr>
<tr>
<td>memory_object_create</td>
<td>352</td>
</tr>
<tr>
<td>memory_object_data_error</td>
<td>117</td>
</tr>
<tr>
<td>memory_object_data_initialize</td>
<td>355</td>
</tr>
<tr>
<td>memory_object_data_provided</td>
<td>119</td>
</tr>
<tr>
<td>memory_object_data_request</td>
<td>121</td>
</tr>
<tr>
<td>memory_object_data_return</td>
<td>123</td>
</tr>
<tr>
<td>memory_object_data_supply</td>
<td>125</td>
</tr>
<tr>
<td>memory_object_data_unavailable</td>
<td>128</td>
</tr>
<tr>
<td>memory_object_data_unlock</td>
<td>130</td>
</tr>
<tr>
<td>memory_object_data_write</td>
<td>132</td>
</tr>
<tr>
<td>memory_object_default_server</td>
<td>334</td>
</tr>
<tr>
<td>memory_object_destroy</td>
<td>134</td>
</tr>
<tr>
<td>memory_object_init</td>
<td>138</td>
</tr>
<tr>
<td>memory_object_lock_completed</td>
<td>141</td>
</tr>
<tr>
<td>memory_object_lock_request</td>
<td>143</td>
</tr>
<tr>
<td>memory_object_ready</td>
<td>146</td>
</tr>
<tr>
<td>memory_object_server</td>
<td>336</td>
</tr>
<tr>
<td>memory_object_set_attributes</td>
<td>148</td>
</tr>
<tr>
<td>memory_object_supply_completed</td>
<td>151</td>
</tr>
<tr>
<td>memory_object_terminate</td>
<td>153</td>
</tr>
<tr>
<td>norma_get_device_port</td>
<td>360</td>
</tr>
<tr>
<td>norma_get_host_port</td>
<td>360</td>
</tr>
<tr>
<td>norma_get_host_paging_port</td>
<td>360</td>
</tr>
<tr>
<td>norma_get_host_priv_port</td>
<td>360</td>
</tr>
<tr>
<td>norma_get_nameserver_port</td>
<td>361</td>
</tr>
<tr>
<td>norma_get_special_port</td>
<td>360</td>
</tr>
<tr>
<td>norma_port_location_hint</td>
<td>363</td>
</tr>
<tr>
<td>norma_set_device_port</td>
<td>364</td>
</tr>
<tr>
<td>norma_set_host_paging_port</td>
<td>364</td>
</tr>
<tr>
<td>norma_set_host_priv_port</td>
<td>364</td>
</tr>
<tr>
<td>norma_set_host_port</td>
<td>364</td>
</tr>
<tr>
<td>norma_set_nameserver_port</td>
<td>365</td>
</tr>
<tr>
<td>norma_set_special_port</td>
<td>364</td>
</tr>
<tr>
<td>norma_task_clone</td>
<td>367</td>
</tr>
<tr>
<td>norma_task_create</td>
<td>369</td>
</tr>
<tr>
<td>notify_server</td>
<td>338</td>
</tr>
<tr>
<td>processor_assign</td>
<td>236</td>
</tr>
<tr>
<td>processor_basic_info</td>
<td>418</td>
</tr>
<tr>
<td>processor_control</td>
<td>238</td>
</tr>
<tr>
<td>processor_exit</td>
<td>240</td>
</tr>
<tr>
<td>processor_get_assignment</td>
<td>241</td>
</tr>
<tr>
<td>processor_info</td>
<td>242</td>
</tr>
<tr>
<td>processor_set_basic_info</td>
<td>419</td>
</tr>
<tr>
<td>processor_set_create</td>
<td>244</td>
</tr>
<tr>
<td>processor_set_default</td>
<td>246</td>
</tr>
<tr>
<td>processor_set_destroy</td>
<td>247</td>
</tr>
<tr>
<td>processor_set_info</td>
<td>248</td>
</tr>
<tr>
<td>processor_set_max_priority</td>
<td>250</td>
</tr>
<tr>
<td>processor_set_policy_disable</td>
<td>252</td>
</tr>
<tr>
<td>processor_set_policy_enable</td>
<td>254</td>
</tr>
<tr>
<td>processor_set_sched_info</td>
<td>420</td>
</tr>
<tr>
<td>processor_set_tasks</td>
<td>255</td>
</tr>
<tr>
<td>processor_set_threads</td>
<td>256</td>
</tr>
<tr>
<td>processor_start</td>
<td>257</td>
</tr>
<tr>
<td>prof_server</td>
<td>340</td>
</tr>
<tr>
<td>receive_samples</td>
<td>160</td>
</tr>
<tr>
<td>sampled_pc</td>
<td>421</td>
</tr>
<tr>
<td>security_id_t</td>
<td>422</td>
</tr>
<tr>
<td>seqnos_default_pager_info</td>
<td>348</td>
</tr>
<tr>
<td>seqnos_default_pager_object_create</td>
<td>350</td>
</tr>
<tr>
<td>seqnos_memory_object_change_completed</td>
<td>112</td>
</tr>
<tr>
<td>seqnos_memory_object_copy</td>
<td>114</td>
</tr>
<tr>
<td>seqnos_memory_object_create</td>
<td>352</td>
</tr>
<tr>
<td>seqnos_memory_object_data_initialize</td>
<td>355</td>
</tr>
<tr>
<td>seqnos_memory_object_data_request</td>
<td>121</td>
</tr>
<tr>
<td>seqnos_memory_object_data_return</td>
<td>123</td>
</tr>
<tr>
<td>seqnos_memory_object_data_unlock</td>
<td>130</td>
</tr>
<tr>
<td>seqnos_memory_object_data_write</td>
<td>132</td>
</tr>
<tr>
<td>seqnos_memory_object_default_server</td>
<td>341</td>
</tr>
<tr>
<td>seqnos_memory_object_init</td>
<td>138</td>
</tr>
<tr>
<td>seqnos_memory_object_lock_complete</td>
<td>141</td>
</tr>
<tr>
<td>seqnos_memory_object_server</td>
<td>343</td>
</tr>
<tr>
<td>seqnos_memory_object_supply_completed</td>
<td>151</td>
</tr>
<tr>
<td>seqnos_memory_object_terminate</td>
<td>153</td>
</tr>
<tr>
<td>seqnos_notify_server</td>
<td>345</td>
</tr>
<tr>
<td>swtch</td>
<td>161</td>
</tr>
<tr>
<td>swtch_pri</td>
<td>162</td>
</tr>
</tbody>
</table>
# Interface and Structure Index

<table>
<thead>
<tr>
<th>Task Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>task_assign</td>
<td>258</td>
</tr>
<tr>
<td>task_assign_default</td>
<td>260</td>
</tr>
<tr>
<td>task_basic_info</td>
<td>420</td>
</tr>
<tr>
<td>task_basic_secure_info</td>
<td>424</td>
</tr>
<tr>
<td>task_change_sid</td>
<td>194</td>
</tr>
<tr>
<td>task_create/task_create_secure</td>
<td>195</td>
</tr>
<tr>
<td>task_get_assignment</td>
<td>262</td>
</tr>
<tr>
<td>task_get_assign</td>
<td>267</td>
</tr>
<tr>
<td>task_get_bootstrap_port</td>
<td>200</td>
</tr>
<tr>
<td>task_get_emulation_vector</td>
<td>198</td>
</tr>
<tr>
<td>task_get_exception_port</td>
<td>200</td>
</tr>
<tr>
<td>task_get_kernel_port</td>
<td>200</td>
</tr>
<tr>
<td>task_get_special_port</td>
<td>200</td>
</tr>
<tr>
<td>task_info</td>
<td>202</td>
</tr>
<tr>
<td>task_priority</td>
<td>263</td>
</tr>
<tr>
<td>task_resume</td>
<td>204</td>
</tr>
<tr>
<td>task_sample</td>
<td>205</td>
</tr>
<tr>
<td>task_set_bootstrap_port</td>
<td>211</td>
</tr>
<tr>
<td>task_set_child_node</td>
<td>371</td>
</tr>
<tr>
<td>task_set_emulation</td>
<td>207</td>
</tr>
<tr>
<td>task_set_emulation_vector</td>
<td>209</td>
</tr>
<tr>
<td>task_set_exception_port</td>
<td>211</td>
</tr>
<tr>
<td>task_set_kernel_port</td>
<td>211</td>
</tr>
<tr>
<td>task_set_special_port</td>
<td>211</td>
</tr>
<tr>
<td>task_suspend</td>
<td>213</td>
</tr>
<tr>
<td>task_terminate</td>
<td>214</td>
</tr>
<tr>
<td>task_thread_times_info</td>
<td>426</td>
</tr>
<tr>
<td>task_threads</td>
<td>215</td>
</tr>
<tr>
<td>thread_abort</td>
<td>164</td>
</tr>
<tr>
<td>thread_assign</td>
<td>265</td>
</tr>
<tr>
<td>thread_assign_default</td>
<td>266</td>
</tr>
<tr>
<td>thread_basic_info</td>
<td>427</td>
</tr>
<tr>
<td>thread_create/thread_create_secure</td>
<td>166</td>
</tr>
<tr>
<td>thread_depress_abort</td>
<td>168</td>
</tr>
<tr>
<td>thread_get_assignment</td>
<td>267</td>
</tr>
<tr>
<td>thread_get_exception_port</td>
<td>169</td>
</tr>
<tr>
<td>thread_get_kernel_port</td>
<td>169</td>
</tr>
<tr>
<td>thread_get_special_port</td>
<td>169</td>
</tr>
<tr>
<td>thread_get_state</td>
<td>171</td>
</tr>
<tr>
<td>thread_info</td>
<td>173</td>
</tr>
<tr>
<td>thread_max_priority</td>
<td>268</td>
</tr>
<tr>
<td>thread_policy</td>
<td>270</td>
</tr>
<tr>
<td>thread_priority</td>
<td>271</td>
</tr>
<tr>
<td>thread_resume/thread_resume_secure</td>
<td>175</td>
</tr>
<tr>
<td>thread_sample</td>
<td>176</td>
</tr>
<tr>
<td>thread_sched_info</td>
<td>429</td>
</tr>
<tr>
<td>thread_set_exception_port</td>
<td>178</td>
</tr>
<tr>
<td>thread_set_kernel_port</td>
<td>178</td>
</tr>
<tr>
<td>thread_set_special_port</td>
<td>178</td>
</tr>
<tr>
<td>thread_set_state/thread_set_state_secure</td>
<td>180</td>
</tr>
<tr>
<td>thread_suspend</td>
<td>182</td>
</tr>
<tr>
<td>thread_switch</td>
<td>183</td>
</tr>
<tr>
<td>thread_terminate</td>
<td>185</td>
</tr>
<tr>
<td>thread_wire</td>
<td>186</td>
</tr>
<tr>
<td>time_value</td>
<td>431</td>
</tr>
<tr>
<td>vm_allocate/vm_allocate_secure</td>
<td>82</td>
</tr>
<tr>
<td>vm_copy</td>
<td>85</td>
</tr>
<tr>
<td>vm_deallocate</td>
<td>87</td>
</tr>
<tr>
<td>vm_inherit</td>
<td>89</td>
</tr>
<tr>
<td>vm_machine_attribute</td>
<td>91</td>
</tr>
<tr>
<td>vm_map</td>
<td>93</td>
</tr>
<tr>
<td>vm_protect</td>
<td>97</td>
</tr>
<tr>
<td>vm_read</td>
<td>99</td>
</tr>
<tr>
<td>vm_region/vm_region_secure</td>
<td>101</td>
</tr>
<tr>
<td>vm_set_default_memory_manager</td>
<td>357</td>
</tr>
<tr>
<td>vm_statistics</td>
<td>104</td>
</tr>
<tr>
<td>vm_statistics</td>
<td>432</td>
</tr>
<tr>
<td>vm_wire</td>
<td>105</td>
</tr>
<tr>
<td>vm_write</td>
<td>107</td>
</tr>
<tr>
<td>vm_region</td>
<td>105</td>
</tr>
<tr>
<td>vm_write</td>
<td>107</td>
</tr>
</tbody>
</table>