MK++ Kernel Interfaces

Version 2.1

November 17, 1997

Revision History:

Revision 1	November 19, 1993	
Revision 1.2	September 15, 1994	Consistency with Kernel Interface
Revision 1.3	May 5, 1995	Consistency with first release
Revision 1.4	November 17, 1995	Formal evaluation version
Revision 2.0	June 12, 1997	Preliminary for MK++V2
Revision 2.1	November 17, 1997	For MK++ V2.1 Release

Copyright 1993-1997 by the The Open Group, Inc.

All rights reserved.

Permission to reproduce this document without fee is hereby granted, provided that the copyright notice and this permission notice appear in all copies, derivative works or modified versions.

The Open Group MAKES NO WARRRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The Open Group shall not be liable for errors contained herein or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.

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

Contents

CHAPTER 1 **Introduction** 1
Interface Descriptions 1
Interface Types 2
Special Forms 2
Parameter Types 3
Error Return Values 3
MIG Server Routines 4

CHAPTER 2 **RPC Interface** 5
mk_rpc 6
mk_rpc_reply 13
rpc_demultiplex 15
server_dispatch *17*
*subsystem*_server 19

CHAPTER 3 **Port Manipulation Interface** 21
port_allocate 22
port_destroy 24
port_extract_right 25
port_get_attributes 26
port_get_refs 28
port_insert_right 30
port_mod_refs 31
port_move 33
port_request_notification 34
port_set_attributes 36
port_pool_allocate 38
port_pool_destroy 39

CHAPTER 4 **Virtual Memory Interface** 41
host_page_size 42
vm_allocate 43
vm_behavior_set 45
vm_copy 47
vm_deallocate 49
vm_inherit 51
vm_map 53

vm_protect 57
vm_read 59
vm_region 61
vm_reserve 63
vm_unreserve 65
vm_unwire 66
vm_wire 67
vm_write 69

CHAPTER 5 **External Memory Management Functions** 71
memory_object_change_attributes 72
memory_object_data_error 74
memory_object_data_supply 76
memory_object_data_unavailable 78
memory_object_destroy 79
memory_object_establish 80
memory_object_get_attributes 82
memory_object_lock_request 84
memory_object_representative 87
memory_object_revoke 90

CHAPTER 6 **Thread Interface** 91
mach_thread_self 92
thread_create 93
thread_create_empty 94
thread_create_running 96
thread_depress_abort 98
thread_get_exception_ports 99
thread_get_special_port 102
thread_get_state 104
thread_info 105
thread_notify 107
thread_resume 108
thread_sample 109
thread_set_exception_ports 110
thread_set_special_port 112
thread_set_state 113
thread_suspend 114
thread_swap_exception_ports 116
thread_switch 119
thread_terminate 121

CHAPTER 7 **Task Interface** 123
mach_task_self 124
task_create 125
task_get_exception_ports 127
task_get_special_port 130
task_info 132
task_resume 134
task_sample 135
task_set_exception_ports 136
task_set_special_port 138
task_suspend 140
task_swap_exception_ports 141
task_terminate 144
task_threads 145

CHAPTER 8 **Host, Clock and Alarm Interface** 147
alarm_create 148
alarm_cancel 150
alarm_set 151
alarm_terminate 153
clock_get_attributes 154
clock_get_time 156
clock_map_time 157
clock_sleep 158
get_time_stamp 160
host_get_clock_service 161
host_info 163
host_kernel_version 165
mach_host_self 166

CHAPTER 9 **Kernel Device Interface** 167
device_close 168
device_get_status 169
device_map 171
device_packet_request 173
device_read 175
device_read_packet 177
device_read_request 179
device_set_filter 181
device_set_status 185
device_write 187

device_write_request 189

CHAPTER 10 **Synchronizers** 191
cv_broadcast 192
cv_create 193
cv_destroy 194
cv_signal 195
cv_wait 196
cv_wait_timed 197
lock_acquire 198
lock_create 199
lock_destroy 200
lock_release 201
lock_try 202
semaphore_create 203
semaphore_destroy 204
semaphore_read_value 205
semaphore_signal 206
semaphore_wait 207
semaphore_wait_timed 208

CHAPTER 11 **Service Interfaces** 209
alarm_notify 210
catch_exception_raise 212
device_packet_complete 215
device_read_complete 217
device_write_complete 218
do_mach_notify_dead_name 219
end_samples 220
receive_samples 221

CHAPTER 12 **External Memory Management Service Interfaces** 223
memory_object_change_completed 224
memory_object_data_request 226
memory_object_data_return 228
memory_object_data_unlock 230
memory_object_lock_completed 232
memory_object_notify 234
memory_object_rejected 237
memory_object_supply_completed 239
memory_object_terminate 241

CHAPTER 13 **Privileged Interfaces** 243
add_partition 244
bootstrap_arguments 245
bootstrap_ports. 246
device_open 248
host_get_boot_info 250
host_get_clock_control 251
host_processor_set_priv. 253
host_reboot. 254
host_sample 255
host_security_id_list 256
host_statistics. 257
ledger_create 258
task_create_security_id 260
thread_wire 262
vm_set_default_memory_manager 263
vm_set_recycle_limit. 264

CHAPTER 14 **Protected Interfaces**. 265
clock_set_attributes 266
clock_set_time 268
ledger_terminate 269
processor_set_policy_control 270
processor_set_statistics 272
processor_set_tasks 273
task_set_policy. 274
thread_set_policy 276

APPENDIX A **Intel Pentium Platform Device Interface Specifics** . 279
Device Names 280
Device Concurrency and Covert Storage Channels. . . 281
Console, Keyboard, and Mouse Drivers 282
Serial Communications (TTY) Drivers 284
VGA Drivers 285
Network Drivers. 286
Floppy Disk Driver 287
SCSI Subsystem. 288

APPENDIX B **Data Structures** 291
host_basic_info 292

host_load_info 294
host_sched_info 295
mapped_tvalspec 296
memory_object_behave_info 297
memory_object_perf_info 298
mk_rpc_data 299
policy_fifo_info 301
policy_rr_info 303
policy_timeshare_info 305
port_attr_engines 307
port_attr_name 308
port_attr_status 309
processor_set_basic_info 310
processor_set_load_info 311
task_basic_info 312
task_thread_times_info 314
thread_basic_info 315
tvalspec 317
vm_region_basic_info 318
vm_statistics 320

APPENDIX C **Error Return Values** 323
Error Code Format 324
MIG Stub Errors 325
RPC Errors 326
Generic Kernel Errors 328
Port Manipulation Errors 329
Virtual Memory Manipulation Errors 330
Miscellaneous Kernel Errors 331
Kernel Device Errors 333

APPENDIX D **Interface and Structure Index** 335

APPENDIX E **Object Index** 339

CHAPTER 1

Introduction

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

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

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

Interface Descriptions

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

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

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

Interface Types

Most of the interfaces in this book are MIG generated interfaces. That is, they are stub routines generated from MIG interface description files. Calling these interfaces will actually invoke an MK++ RPC on the port specified by the first argument in the call. This has two important effects.

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

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

Most of these interfaces are of the type **Function**. This means that there is actually a C callable function (most likely in **libmach.a**) that has the calling signature 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 notification RPCs invoked by the kernel. That is, it is assumed that some task is prepared to handle RPCs l on a port to which the kernel is to direct RPCs. A received notification RPC will be passed to a MIG generated server routine (*service*_**server**) that will call an appropriate server target function. It is these server target functions, one for each different RPC that the kernel generates, that are listed as **Server Interface**s.

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.

Parameter Types

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

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

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

If the notation says “structure”, the parameter is a direct structure type whose layout is described in an appendix to this document.

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.

Error Return Values

An appendix to this document describes the various error return values defined by the kernel. However, since the kernel interfaces are actually MIG generated stubs that invoke RPCs, 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 **mk_rpc**. As such, the various RPC errors (MK_RPC_...) are possible.

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

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

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

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

MIG Server Routines

For each logical set of RPCs sent from the kernel to server interfaces, there is one or more MIG-generated server routines used to parse the RPC and call the appropriate server routine. The currently supplied routines are the following. Each server interface description names the corresponding MIG server routine.

- **alarm_notify_server**— Handles kernel generated clock alarms.
- **device_completion_server**— Handles messages from a kernel device driver
- **exception_server**— Handles kernel messages for an exception handler
- **memory_cache_notif_server**— Handles kernel messages for user pagers
- **memory_object_notif_server**— Handles kernel messages for the default memory manager
- **notify_server**— Handles kernel generated RPC notifications
- **sample_server**— Handles kernel generated PC sample messages

The general interface to a MIG generated server routine is described in the ***subsystem*_server** description.

CHAPTER 2 RPC Interface

This chapter discusses the kernel's Remote Procedure Call (RPC) interface. The interfaces described here are the only ones directly involved in invoking and providing service for remote procedure calls.

mk_rpc

System Trap / Function — Invokes a synchronous RPC

SYNOPSIS

```
kern_return_t mk_rpc
        (mach_port_t        target_port,
        natural_t           method_id,
        mk_rpc_data_t*      invocation_buffer,
        mk_rpc_size_t       invocation_buffer_size);

kern_return_t mk_rpc_003_003
        (mach_port_t        target_port,
        natural_t           method_id,
        natural_t           arg1,
        natural_t           arg2,
        natural_t           arg3,
        natural_t           output_args[3]);
```

DESCRIPTION

The **mk_rpc** system trap invokes a synchronous remote procedure call (RPC) on *target_port*. The data passed to the target server is contained in *invocation_buffer*, and may contain port rights and addresses of regions of separately allocated memory, as well as untyped user data. The return value indicates whether problems were found, with the ports or data supplied, while performing the RPC.

The **mk_rpc_003_003** system trap provides an optimized calling signature for use when the RPC involves transfer of small amounts of data but no port rights or out-of-line memory regions. Up to 3 scalar arguments and 3 scalar return values may be passed. In this case, the kernel need not copy in the invocation buffer and analyze it for ports or out-of-line memory regions.

PARAMETERS

target_port
: [in send right] Specifies the port that is the target of this RPC.

method_id
: [in scalar] This scalar indicates the method to be invoked by the server.

invocation_buffer
: [pointer to in structure] An RPC buffer describing and containing the data to be passed along with the RPC invocation. This must be naturally aligned. The format of this structure is described below.

invocation_buffer_size
[in scalar] The size of the invocation buffer in bytes.

NOTES

The MK++ kernel provides a capability-based form of inter-process communication based upon synchronous remote procedure calls (RPC). The RPC primitives efficiently support many different styles of interaction, including object-oriented distributed programming, reliably acknowledged one-way messaging, and sending of arbitrary amounts of data. Conceptually, all data is copied during RPC invocation and reply.

Major Concepts

RPC is the unifying abstraction used for all transfers of data and control between user tasks and between a user task and the kernel. RPC is the mechanism by which servers provide services to clients. A client requesting a service is said to invoke an RPC. An RPC invocation is synchronous: the client blocks until the server completely processes the request, explicitly returning control to the client. Once control has returned to the client all data transfer has taken place, and there is no lingering dependency on the server.

The RPC primitives let tasks invoke or service remote procedure calls. Tasks invoke RPCs over ports. An RPC transfers control to the task servicing the port, along with an arbitrary collection of data and port rights. Likewise, both data and port rights may be returned to the invoking task along with the reply. This passing of port rights with both invocations and replies is very significant, since receiving ports in an RPC is the primary way that a task obtains new rights. The data passed may include untyped data and out-of-line (OOL) memory regions of arbitrary size, along with OOL descriptors indicating where any OOL data in the server's reply should be deposited. When a task services an RPC containing such out-of-line regions of data, all of the data must fit into the receive buffer of the empty thread that is about to provide the service. When that server replies with OOL data, the amount of data must not exceed the size of the OOL areas indicated in the invocation request (the reply size may be less than the area supplied by the invoker without error). Note that it is not an error for the server to provide data for some but not all of the OOL areas.

A port provides a communication channel between a client requesting a service and a server that provides it. A port has a single receiver and (potentially) multiple senders. A port that represents a kernel-supported resource has the kernel as the receiver. A port that names a service provided by a task has that task as the port's receiver. Receipt of requests/invocations is performed only upon port pools; while ports may be moved in and out of port pools, receive rights may not be transferred between tasks, and so receivership cannot change. Ports mediate requests for service, by affording a channel to the provider of the service. Tasks operate on a port to invoke or provide service by exercising capabilities (rights) for the port. Multiple tasks can hold send rights for a port. Only one task can hold the receive capability (receive right) for a port. Send rights can be transferred between tasks via RPC.

Tasks can service RPC invocations only via port pools. The port pool abstraction allows one or more empty threads to wait for an RPC invocation from any of several ports. Empty threads are the recipients of RPC invocations, and can only execute when they have been loaned some resources from the thread making the RPC request. Tasks manipulate port pools with a port pool name, which is taken from the same name space as are the port rights. The port pool name may not be transferred via RPC. A port pool holds receive rights. A thread invoking requests targeted at a port in a port pool with no available empty threads (either because all are currently in use or because none are currently associated with the port pool) will be blocked. Invoking threads are queued for service by scheduling priority. When an empty thread becomes available, the most eligible blocked invoking thread donates the necessary resources to the empty thread which then executes the requested service.

Port rights are a secure, location-independent way of naming ports. The port is a protected data structure providing a logical channel between tasks, only accessible via the kernel's exported 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 invoke service over a port to which it shouldn't have access. Port rights do not carry any location information. When a send right for a port moves from task to task, perhaps between tasks on different nodes, the receiver remains unaware of this change, and RPC service continues to function as before. There is a test for port locality exported by the kernel; it is used by the runtime stubs to determine when fully self-describing buffers must be used. Stubs which receive fully self-describing buffers must reply with fully self-describing buffers. There are a small number of architecture-specific optimized signatures that are not fully self-describing; these provide more efficient RPC for certain common calling sequences that don't require the full generality of **mk_rpc**. These formats are described elsewhere.

Port Rights

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

There are two basic kinds of rights: receive rights and send rights. A port name can name either or both of these types of rights, name a port pool, or name nothing.

A port is destroyed, or dies, when its receive right is deallocated or its ledger is destroyed. When a port dies, send rights for the port are not affected, although they can no longer be used to invoke an RPC. Any pending RPCs awaiting service threads at the port return an error.

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

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

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

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

Memory

An RPC can transmit untyped, marshalled, user data "in-line," or it can describe one or more regions of the invoker's address space that are to be transferred "out-of-line." In this case, the RPC is accompanied by a logical copy of the memory. The kernel will transfer such OOL regions in the most efficient way it can, given the destination and the type of underlying transport in use. The invoker of the RPC must explicitly request an out-of-line transfer. Such a region is described as an arbitrary region of the sender's address space. The sender sees this memory as being copied to the receiver, and all such regions are to be considered volatile until the RPC returns.

All data that is transmitted with an RPC is deposited into the receive buffer of the empty thread that is about to service the RPC. Out-of-line regions are deposited as out-of-line descriptors which point into unused portions of the receive buffer. Thus the receive buffer must be large enough to contain the header and all data for the largest RPC request that the server is able to handle. RPC provides no automatic VM manipulations; receive buffers are pre-allocated, and there is no need to detect and free memory which has been automatically created in the server's address space.

Null out-of-line memory is legal. If the out-of-line region size is zero, then the region's specified address is ignored.

To reply to the client after servicing an RPC request, the server supplies the data to be transferred along as control moves back to the client. While the local stack or the receive buffer are the ideal places to assemble this data, it may be necessary to allocate a reply buffer that is too large to comfortably fit on the stack. In this case, the buffer must be considered a per-empty-thread object, since it is not safe to free this buffer until the next invocation using that empty thread takes

place. In other words, there is no point at which that thread executes after the kernel has safely transmitted the reply data.

RPC Invocation

When a thread invokes an RPC, the argument data is marshalled, and travels along with the thread's shuttle as it migrates into the task providing the service. This shuttle is coupled with an empty thread from the server's port pool, forming a complete, runnable, thread which goes about performing the needed service. RPC invocation is synchronous and reliable. This means that the operation will either be performed completely or fail cleanly, without visible side-effects. Any data passed is volatile; this means that the results are undefined if another thread should overwrite any data being used by an RPC while that RPC is in progress. Only when an RPC has returned can a thread safely modify the data that was used as input to the RPC. Included in the input data for **mk_rpc** is a pointer/length pair describing the buffer which is to hold any reply data from the server, plus OOL descriptors indicating where OOL reply data should be placed. The input data format was designed to allow the reply buffer to overlay the input buffer. In the event of an RPC failure, the contents of the reply buffer and any OOL areas are undefined.

RPC Service

Servers provide for RPC handling by populating a port pool with an appropriate number of empty threads, which, along with the shuttles donated by invoking threads, perform the requested services. These empty threads require an area of memory to use as a user stack, and an area of memory to hold the invoker's data during the servicing of the RPC. These newly-whole threads begin running at the routine used to dispatch incoming RPCs with an indication of the port requesting service, an indication of the service requested, and a pointer to any data passed. After performing the service, the thread explicitly traps back into the kernel, providing the data to be returned. The shuttle is returned to the requesting thread along with any reply data, and the empty thread is put back on the port pool to await the next RPC request.

Details of OOL Data Handling

While in principle both scatter (output to distinct areas of memory) and gather (input from distinct areas of memory) could be supported on both the client and server sides of an RPC operation, **mk_rpc** supports all except server-side scatter.

In an RPC invocation, client side gather is specified by the OOL descriptors in the invocation buffer. These regions will be transferred to the server's receive buffer without change in alignment. The implementation is free to perform alignment individually on the end of the in-line region and the start and end of each out-of-line region; this means that for an alignment of granularity N, (N - 1) * (1 + 2 * (# of OOL regions)) bytes may be used for padding in the server receive buffer. In some cases, this uses more memory than is strictly necessary. The contents of all such padding memory are undefined. The alignment granularity for an implementation is available as the constant MK_RPC_ALIGNMENT; this constant need not match the word size or the

size of any data type on the machine, and may vary based on processor architecture.

In the RPC reply, server side gather is specified by the OOL descriptors in the server's reply buffer, and client side scatter is implemented via a second set of OOL descriptors in the client's invocation buffer. An exact match in number of OOL descriptors is required; each OOL region of the server's reply must fit within the client's corresponding region (the sizes need not match precisely). OOL regions are not appended to, split, nor coalesced. Mismatches in the number of OOL descriptors or overruns in their sizes cause the RPC to abort; receive buffer contents are undefined in this case (partial writes may have occurred). While arbitrary alignment of OOL regions is permitted, programmers are cautioned that the best performance is obtained when the addresses and sizes of OOL regions are zero modulo MK_RPC_ALIGNMENT.

RETURN VALUE

The **mk_rpc** trap can generate the following return codes:

MK_RPC_SUCCESS
: The RPC has completed successfully.

MK_RPC_TOO_SMALL
: The specified *invocation_buffer* was smaller than the minimum size for a mk_rpc_desc_t.

MK_RPC_TOO_LARGE
: The specified *invocation_buffer* was larger than the maximum size than the receiver was willing to accept, or the reply buffer was not large enough to handle the reply.

MK_RPC_BAD_REQUEST
: The RPC header was inconsistent.

MK_RPC_INVALID_DATA
: The address of the supplied invocation buffer was invalid.

MK_RPC_INVALID_BUFF
: The address of the reply buffer was invalid.

MK_RPC_INVALID_NAME
: The supplied *target_port* was invalid or MACH_PORT_NULL.

MK_RPC_NO_MEMORY
: A resource shortage prevented the successful completion of the RPC.

MK_RPC_PORT_DEAD
: The *target_port* was destroyed before the RPC invocation reached it.

MK_RPC_MISMATCH
: Mismatch between request and reply specifications.

MK_RPC_OOL_MISMATCH
: Mismatch of OOL size or number of OOL regions.

MK_RPC_ABORTED
: The operation was terminated in the kernel.

RELATED INFORMATION

Functions: **mk_rpc_reply**, **port_allocate, port_pool_allocate, thread_create_empty**, **port_request_notification**.

Data Structures: **mk_rpc_data**.

mk_rpc_reply

System Trap / Function — Completes a synchronous RPC

SYNOPSIS

```
kern_return_t mk_rpc_reply
        (mk_rpc_data_t*          reply_buffer,
        mk_rpc_size_t            reply_buffer_size);

kern_return_t mk_rpc_reply_buff
        (mk_rpc_data_t*          reply_buffer,
        mk_rpc_size_t            reply_buffer_size,
        mk_rpc_data_t*           new_buffer,
        mk_rpc_size_t            new_buffer_size);

kern_return_t mk_rpc_reply_003
        (natural_t               rval1,
        natural_t                rval2,
        natural_t                rval3);
```

DESCRIPTION

The **mk_rpc_reply** trap is called by a server to return control to the invoking thread. The data returned with this reply is contained in *reply_buffer*, which is formatted similarly to the *invocation_buffer*. These structures have been designed to allow (but not require) the *reply_buffer* to overlay the *invocation_buffer*. While this call does not normally return; attempting to call **mk_rpc_reply** from a thread that is not servicing an RPC will fail with KERN_INVALID_VALUE.

The **mk_rpc_reply_buff** trap is similar to **mk_rpc_reply**, but allows the server to install a new buffer for receipt of data for the next incoming RPC. This avoids the need to copy the data out of the *client_data* for the current invocation. The server is responsible for handling the eventual deallocation of *client_data*.

PARAMETERS

reply_buffer

[in pointer to structure] A naturally aligned buffer containing the data to be returned to the invoking thread. A null reply buffer results in no data being returned with the reply. Reply OOL descriptors in the *reply_buffer* are ignored.

reply_buffer_size

[in scalar] The size of the reply buffer in bytes. Zero should be supplied if no reply buffer is given.

NOTES

RETURN VALUE

The **mk_rpc_reply** trap does not return if called from an RPC service thread.

RELATED INFORMATION

Functions: **mk_rpc, server_dispatch**

Data Structures: **mk_rpc_data**

rpc_demultiplex

Server Interface— Service a synchronous RPC

SYNOPSIS

void **rpc_demultiplex**
(natural_t *opaque1,*
natural_t *opaque2*,
natural_t *method_id,*
mk_rpc_data_t* *client_data,*
security_id_t* *security_id*);

DESCRIPTION

The **rpc_demultiplex** function is invoked by **_server_dispatch**, which translates the machine-dependent invocation to a C calling sequence. This routine is responsible for locating the correct server routine, and responding to the RPC.

PARAMETERS

method_id
[in scalar] This scalar indicates the method to be invoked by the server.

opaque1
[in scalar] This scalar is passed to the dispatch routine to indicate the port that is the target of this RPC invocation. This ID is an attribute of the receive right, established when the port is created with **port_allocate**. It defaults to the name of the receive right, and can be changed using **port_set_attributes**.

opaque2
[in scalar] This scalar is passed to the dispatch routine to indicate additional information about the port that is the target of this RPC invocation. This ID is an attribute of the receive right, established when the port is created with **port_allocate**. It can be changed using **port_set_attributes**.

client_data
[in pointer to structure] A naturally aligned buffer into which the invoking client's data is deposited.

security_id
[in pointer to structure] The security identifier associated with the invoking (client) task. Kernel-initiated RPCs (notifications and exceptions) carry the identifier SECURITY_ID_KERNEL.

NOTES

An example **rpc_demultiplex** function is provided in libmach to dispatch RPC invocations to the appropriate MIG-generated server decode routines.

The libmach version of **rpc_demultiplex** has several assumptions about how the port and empty thread receive buffer were set up that need to be adhered to if this function is to be used:

- The *opaque2* value specified to **port_allocate** must be a function pointer to a MIG subsystem server routine or other routine with the same calling sequence. (see **subsystem_server**)
- The size of the receive buffer (specified to **port_allocate**) must be stored in memory just before the start of the receive buffer.
- The reply buffer can be placed in memory just beyond the end of the receive buffer. The first word of this region, if non-zero, is used limit the size of the reply buffer.

The last conditions imply that a reply with a payload can only be generated if the original request contained a payload, since the address of the receive buffer is only passed to the RPC invocation if there is a a request payload.

In order to allow use with a threading library, such as libcthreads, the libmach version of **rpc_demultiplex** invokes a threading library callback through the global pointer **_thread_rpc_routine** before calling the MIG server.

RETURN VALUE

The **rpc_demultiplex** function should return only if there is a formatting mismatch in the invocation. This function normally calls **mk_rpc_reply** to terminate the invocation and return information and control to the invoker.

RELATED INFORMATION

Functions: **mk_rpc_reply, server_dispatch.**

Data Structures: **mk_rpc_data, security_id_t**.

server_dispatch

Server Interface— Service a synchronous RPC

SYNOPSIS

void ***server_dispatch***();

DESCRIPTION

server_dispatch represents the routine which is invoked within the target server to handle a client's invocation of **mk_rpc**. One dispatch routine is registered per port pool at the time that the port pool is created with **port_pool_allocate**.

It is the purpose of this dispatch function to acquire the invocation information passed by the kernel and call the correct stub routine. The actual calling sequence for the dispatch function is machine-dependent, and may not even be expressable in C. For example, on the Pentium platform, the ***server_dispatch*** routine is invoked with the arguments in processor registers.

PARAMETERS

The information passed to this routine, in a machine-dependent fashion, is summarized below.

method_id

[in scalar] This scalar indicates the method to be invoked by the server.

opaque1

[in scalar] This scalar is passed to the dispatch routine to indicate the port that is the target of this RPC invocation. This ID is an attribute of the receive right, established when the port is created with **port_allocate**. It defaults to the name of the receive right, and can be changed using **port_set_attributes**.

opaque2

[in scalar] This scalar is passed to the dispatch routine to indicate additional information about the port that is the target of this RPC invocation. This ID is an attribute of the receive right, established when the port is created with **port_allocate**. It can be changed using **port_set_attributes**.

client_data

[in pointer to structure] A naturally aligned buffer into which the invoking client's data is deposited. This is the same address as was passed as the receive buffer to **thread_create_empty**

security_id

[in pointer to structure] The security identifier associated with the invoking (client) task. Kernel-initiated RPCs (notifications and exceptions) carry the identifier SECURITY_ID_KERNEL.

NOTES

An example dispatch function suitable for use with MIG-generated servers is provided in **libmach.a**. The **_server_dispatch** routine can be specified as the dispatch address to **port_allocate**. When invoked, this routine pushes the arguments and calls the C function **rpc_demultiplex** to locate the correct server routine. If **rpc_demultiplex** returns, then **_server_dispatch** calls **_consume_buffer** to remove any port rights obtained due to the invocation, and then replies with an error.

The **_server_dispatch** function in libmach does not accept optimized (**mk_rpc_003_003**) call signature invocations.

The **rpc_demultiplex** and **_consume_buffer** routines may be provided by the user, otherwise the libmach implementation is used.

RETURN VALUE

The **server_dispatch** function must not return, since it is at the top of a call stack. This function must call **mk_rpc_reply** to terminate the invocation.

RELATED INFORMATION

Functions: **mk_rpc, mk_rpc_reply, rpc_demultiplex.**

Data Structures: **mk_rpc_data, security_id_t**.

*subsystem*_server

Server Interface— Decode and dispatch a synchronous RPC

SYNOPSIS

```
boolean_t subsystem_server(
        mach_port_t                 opaque1,
        natural_t                   method_id,
        mk_rpc_data_t *             client_data,
        mk_rpc_data_t **            reply_data,
        security_id_t *             security_id,
        kern_return_t *             retcode);
```

DESCRIPTION

***subsystem*_server** represents the MIG generated server decode routine which is invoked within the target server to handle a client's invocation of **mk_rpc** on a particular port. The routine will return TRUE if the *method_id* matched a method for this server, FALSE otherwise.

The ***subsystem*_server** routine can be used as the *opaque2* argument to **port_allocate**, or a routine can be specified that iterates over multiple different ***subsystem*_server** routines for a port that services more than one type of rquest.

PARAMETERS

opaque1

[in scalar] This scalar is passed to the dispatch routine to indicate the port that is the target of this RPC invocation. This ID is an attribute of the receive right, established when the port is created with **port_allocate**. It defaults to the name of the receive right, and can be changed using **port_set_attributes**.

method_id

[in scalar] This scalar indicates the method to be invoked by the server.

client_data

[in pointer to structure] A naturally aligned buffer into which the invoking client's data is deposited. This is the same address as was passed as the receive buffer to **thread_create_empty**

reply_data

[in/out pointer to structure] A naturally aligned buffer into which the server's reply data is deposited. To return scalar return code, the server sets **reply_data* to NULL, and sets **retcode* to the error return.

security_id

[in pointer to structure] The security identifier associated with the invoking (client) task. Kernel-initiated RPCs (notifications and exceptions) carry the identifier SECURITY_ID_KERNEL.

retcode

[out scalar] The return code from called the RPC server stub, or the argument decoding routine, if no reply is specified in **reply_data.*

NOTES

This entry represents the calling signature and behavior of the MIG-generated server decode routines. The ***subsystem*_server** function pointer can be used as the *opaque2* value to **port_allocate** if the empty thread dispatch address is set to the **libmach.a** routine **_server_dispatch**.

RETURN VALUE

The function return value should be interpreted as follows

TRUE

The *method_id* matched a method handled by this subsystem

FALSE

The *method_id* did not match a method handled by this subsystem

If the function returns TRUE, the **reply_data* value is an RPC payload result to be returned to the client.

If **reply_data* is NULL, a scalar return value in **retcode* should be returned to the client.

RELATED INFORMATION

Functions: **server_dispatch, rpc_demultiplex.**

Data Structures: **mk_rpc_data, security_id_t**.

CHAPTER 3

Port Manipulation Interface

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

port_allocate

Function — Creates a port

SYNOPSIS

```
kern_return_t port_allocate
        (mach_port_t            space,
        port_type_t             type,
        mach_port_name_t        opaque1,
        mach_port_name_t        opaque2,
        mach_port_name_t        port_pool,
        mach_port_name_t *      name);
```

DESCRIPTION

The **port_allocate** function creates a new receive right in the specified RPC name space (task). The new right's name is returned in *name*. It doesn't have any extant send rights. *space* does not hold send rights for the new port, only the receive right. **port_insert_right** and **port_extract_right** can be used to convert the receive right into a combined send/receive right.

A port is associated with two scalars that are passed to the server's dispatch routine each time an RPC is invoked on the port. These scalars may be set when calling **port_allocate** by setting the *opaque*1 and *opaque*2 parameters. If the caller sets *opaque*1 to 0 (zero), the value passed to the dispatch routine will be the name of the receive right. (The scalars can also be set—and modified—using **port_set_attributes**.)

A port has a scalar associated with it that is passed to the server's dispatch routine upon each RPC invocation over the port. The value of this scalar defaults to the name of the receive right, but may be set using the *server_name* argument, and/or changed using **port_set_attributes**.

In order to receive RPC invocations, the port must belong to a port pool. The port pool may be specified in the *port_pool* argument, or the port can be moved into a pool later using **port_move**.

PARAMETERS

space

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

type

[in scalar] The type of port to be created. This is one of the following:

PORT_TYPE_COMMON

port_allocate creates an ordinary port.

PORT_TYPE_NOTIFICATION

port_allocate creates a notification port. The notification port will have one delivery engine (kernel thread) allocated and associated with it to be used for notification delivery. This number may be changed using **port_set_attributes**.

opaque1, *opaque2*

[in scalar] Values supplied to the server when an RPC is invoked on this port. The contents of these parameters are entirely opaque to the kernel, with one exception: if the caller sets *opaque*1 to 0 (zero), the value passed to the dispatch routine will be the name of the receive right.

port_pool

[in scalar] Port pool with which the newly allocated port should be associated. If a null or invalid name is supplied, the port will not be placed into a port pool.

name

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **port_insert_right, port_extract_right**, **port_mod_refs, port_move**, **port_set_attributes, port_pool_allocate, server_dispatch**.

port_destroy

Function — Removes a task's rights for a name

SYNOPSIS

kern_return_t **port_destroy**
(mach_port_t *space*,
mach_port_name_t *name*);

DESCRIPTION

The **port_destroy** function deallocates all rights denoted by a name. The name becomes immediately available for reuse.

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

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

If *name* denotes a receive right, threads waiting for RPC service for the port will receive an error return. If any send rights naming the port have dead-name requests registered, then dead-name notifications are generated for them.

If *name* denotes a notification port, all delivery engines associated with the port are destroyed.

PARAMETERS

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **port_allocate, port_mod_refs**, **port_request_notification**.

port_extract_right

Function — Extracts a copy of a send right from a task

SYNOPSIS

```
kern_return_t port_extract_right
        (mach_port_t            space,
        mach_port_name_t        name,
        mach_port_t *           port);
```

DESCRIPTION

The **port_extract_right** function produces a copy of a specific send right and returns it to the caller as if the task had sent the right voluntarily in a call to **mk_rpc**. Send rights can only be copied, not moved. Receive rights cannot be copied or moved.

PARAMETERS

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

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

port
: [out send right] The extracted send 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: **port_insert_right**, **mk_rpc**.

port_get_attributes

Function — Returns information about a port

SYNOPSIS

```
kern_return_t port_get_attributes
        (mach_port_t                space,
        mach_port_name_t            name,
        port_info_flavor_t          flavor,
        port_info_t                 port_info_out,
        mk_rpc_size_t *             port_info_outSize);
```

DESCRIPTION

The **port_get_attributes** function returns an information structure of type *flavor*. The caller must hold the port's receive right.

PARAMETERS

space

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

name

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

flavor

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

PORT_ATTR_STATUS

Returns information about the current status of this port; this includes the port pool containing this port, if any, and the type of port this is, notification or common. The declaration of this data is found in structure **port_attr_status**.

PORT_ATTR_ENGINES

Returns the number of concurrent notification deliveries that this port currently supports. The declaration of this data is found in structure **port_attr_engines**. Valid only if *name* is a notification port.

PORT_ATTR_NAME

Returns the opaque values supplied to the server when an RPC is invoked over this port. The declaration of this data is found in structure **port_attr_name**.

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

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

NOTES

This interface is machine word length specific because of the port name parameter and the port pool name in the *port_attr_status* structure returned.

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

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

KERN_INVALID_TYPE
: *name* denoted a right, but the port was the wrong type of port for the given operation.

RELATED INFORMATION

Functions: **port_allocate**, **port_set_attributes**.

Data Structures: **port_attr_status**, **port_attr_engines**, **port_attr_name**.

port_get_refs

Function — Retrieves the number of user references for a right

SYNOPSIS

```
kern_return_t port_get_refs
        (mach_port_t            space,
        mach_port_name_t        name,
        port_right_t            right,
        mach_port_urefs_t *     refs);
```

DESCRIPTION

The **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 that a name may simultaneously denote send and receive rights. The number of references for receive rights is always one.

PARAMETERS

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

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

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

refs
: [out scalar] Number of user references.

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

RELATED INFORMATION

Functions: **port_mod_refs**.

port_insert_right

Function — Inserts a copy of a send right into a task

SYNOPSIS

```
kern_return_t port_insert_right
        (mach_port_t            space,
        mach_port_t             port,
        mach_port_name_t *      name);
```

DESCRIPTION

The **port_insert_right** function inserts a copy of the caller's send right for a port into the target task. The value of *port* may not be MACH_PORT_NULL.

If the target already has send rights for the port, then those send rights gain an additional user reference. Otherwise, the target gains a send right, with a user reference count of one.

PARAMETERS

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

port
: [in send right] The send right.

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *port* did not denote a right.

KERN_INVALID_RIGHT
: *port* denoted a right, but not a send right.

RELATED INFORMATION

Functions: **port_extract_right**, **mk_rpc**.

port_mod_refs

Function — Changes the number of user refs for a right

SYNOPSIS

```
kern_return_t port_mod_refs
        (mach_port_t            space,
        mach_port_name_t        name,
        port_right_t            right,
        mach_port_delta_t       delta);
```

DESCRIPTION

The **port_mod_refs** function is used to modify user reference counts on port rights. The kernel destroys the right if its reference count is set 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: receive rights and the dead-name request may only have one user reference. The resulting number of user references can't be negative. If the resulting number of all user references is zero, the effect is to deallocate the right. There is an implementation-defined maximum number of user references. If the call destroys the right, then the effect is as described for **port_destroy**.

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

PARAMETERS

space

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

name

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

right

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

delta

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

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

RELATED INFORMATION

Functions: **port_destroy**, **port_get_refs**.

port_move

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

SYNOPSIS

```
kern_return_t port_move
        (mach_port_t        space,
        mach_port_name_t    port,
        mach_port_name_t    pool);
```

DESCRIPTION

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

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *port* or *pool* did not denote a right.

KERN_INVALID_RIGHT
: *port* denoted a right, but not a receive right, or *pool* denoted a right, but not a port pool.

RELATED INFORMATION

Functions: **port_get_attributes**.

port_request_notification

Function — Request a notification of a port event

SYNOPSIS

```
kern_return_t port_request_notification
        (mach_port_t            space,
        mach_port_name_t        name,
        mk_rpc_id_t             id,
        mach_port_name_t        notify_name,
        mach_port_t *           previous);
```

DESCRIPTION

The **port_request_notification** function registers a request for a notification and supplies a receive right to the notification port that will be used to receive the notification. It is an atomic swap, returning the previously registered receive right (or MACH_PORT_NULL for none). A notification request may be cancelled by providing MACH_PORT_NULL.

PARAMETERS

space

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

name

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

id

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

MACH_NOTIFY_DEAD_NAME

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

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

notify
: [in notify receive right] A receive right, to which the notification will be sent.

previous
: [out notify receive right] The previously registered 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 an invalid right.

KERN_INVALID_VALUE
: *name* specifies an invalid value.

RELATED INFORMATION

Functions: **port_get_attributes**.

port_set_attributes

Function — Sets attributes of a port

SYNOPSIS

```
kern_return_t port_set_attributes
        (mach_port_t            space,
        mach_port_name_t        name,
        port_info_flavor_t      flavor,
        port_info_t             port_info,
        mk_rpc_size_t           port_infoSize);
```

DESCRIPTION

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

PARAMETERS

space

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

name

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

flavor

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

PORT_ATTR_ENGINES

Sets the number of concurrent notifications that this can be supported on this port. The declaration of this data is found in structure **port_attr_engines**, comprised of a single integer. The default value is one. This value may be raised or lowered using this interface, which is only valid for notification ports. Specifying a negative number of delivery engines causes that number of delivery engines to be deleted, if possible; the actual number deleted is returned in place in the **port_attr_engines** structure.

PORT_ATTR_NAME

Sets the scalar values delivered to the server when an RPC is invoked over this port. The declaration of this data is found in structure **port_attr_name**, comprised of a two scalars. The default value of the first scalar is the name of the receive right.

port_info

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

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

KERN_INVALID_TYPE
: *name* denoted a right, but the port was the wrong type of port for the given operation.

KERN_INVALID_ARGUMENT
: The caller provided an invalid *flavor* or did not provide a sufficiently-sized *port_infosize*.

RELATED INFORMATION

Functions: **port_allocate**, **port_get_attributes**.

Data Structures: **port_attr_engines**, **port_attr_name**.

port_pool_allocate

Function — Creates a port pool

SYNOPSIS

```
kern_return_t port_pool_allocate
        (mach_port_t                    space,
        mach_port_name_t *              name);
```

DESCRIPTION

The **port_pool_allocate** function creates a new port pool in the specified task. The new port pool's name is returned in *name*. The new port pool has no members, nor does it have any associated empty threads. The scheduling policy mask associated with the port pool initially allows threads of any policy to invoke RPCs; this may be changed with **port_pool_set_attributes**.

PARAMETERS

space

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

name

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **port_pool_destroy, port_move**, **port_pool_set_attributes**.

port_pool_destroy

Function — Removes a port pool from a task

SYNOPSIS

kern_return_t **port_pool_destroy**
(mach_port_t *space*,
mach_port_name_t *name*);

DESCRIPTION

The **port_pool_destroy** function deallocates the port pool denoted by *name*. The name becomes immediately available for reuse.

All receive rights in the port pool are implicitly removed. Threads waiting for RPC service will be transferred to a queue associated with the destination port, and will remain blocked until the port is moved back into a port pool. Empty threads associated with the port pool that are idle will be terminated. Empty threads that are servicing RPC requests will be marked so that they will be terminated when they return back to the kernel.

PARAMETERS

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a port pool.

RELATED INFORMATION

Functions: **port_pool_allocate**.

CHAPTER 4

Virtual Memory Interface

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

host_page_size

Function — Return the specified host's page size

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_host_self**.

vm_allocate

Function — Allocates a region of virtual memory

SYNOPSIS

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

DESCRIPTION

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

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

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

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

PARAMETERS

task

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

address

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

size

[in scalar] The number of bytes to allocate.

where

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

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

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

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

VM_ALLOCATE_WIRED
: The kernel marks the allocated pages wired.

NOTES

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

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

RETURN VALUE

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

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

RELATED INFORMATION

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

vm_behavior_set

Function — Sets expected data access hints

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

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

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

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

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

vm_copy

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

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

NOTES

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

RETURN VALUE

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

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

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

RELATED INFORMATION

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

vm_deallocate

Function — Deallocates a region of virtual memory

SYNOPSIS

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

DESCRIPTION

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

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

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

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

vm_inherit

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

PARAMETERS

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

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

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

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

  VM_INHERIT_SHARE
  : Allows child tasks to share the region.

VM_INHERIT_COPY
: Gives child tasks a copy of the region.

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

vm_map

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

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

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

PARAMETERS

task

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

address

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

size

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

mask

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

where

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

VM_ALLOCATE_FIXED

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

VM_ALLOCATE_RESERVED

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

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

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

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

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

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

  VM_PROT_READ
  : Allows read access.

  VM_PROT_WRITE
  : Allows write access.

  VM_PROT_EXECUTE
  : Allows execute access.

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

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

  VM_INHERIT_SHARE
  : Allows child tasks to share the region.

  VM_INHERIT_COPY
  : Gives child tasks a copy of the region.

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

NOTES

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

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

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

CAUTIONS

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

RETURN VALUE

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

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

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

RELATED INFORMATION

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

vm_protect

Function — Sets access privileges for a region of virtual memory

SYNOPSIS

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

DESCRIPTION

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

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

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

Protection cannot be set for regions reserved but not allocated.

PARAMETERS

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

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

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

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

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

  VM_PROT_READ
  : Allows read access.

  VM_PROT_WRITE
  : Allows write access.

  VM_PROT_EXECUTE
  : Allows execute access.

NOTES

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

RETURN VALUE

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

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

RELATED INFORMATION

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

vm_read

Function — Reads a task's virtual memory

SYNOPSIS

```
kern_return_t vm_read
        (mach_port_t        task,
        vm_address_t        address,
        vm_address_t        destination,
        mk_rpc_size_t       destination_size);
```

DESCRIPTION

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

PARAMETERS

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

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

destination
: [in scalar] The address within the calling task to which the data is to be read.

destination_size
: [in scalar] The size, in bytes, of the region to be read.

NOTES

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

RETURN VALUE

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

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

KERN_MEMORY_ERROR
: Some portion of the range is unallocated or inaccessible.

RELATED INFORMATION

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

vm_region

Function — Returns information on a region of virtual memory

SYNOPSIS

```
kern_return_t vm_region
        (mach_port_t                target_task,
        vm_address_t *              address,
        vm_size_t *                 size,
        vm_region_flavor_t          flavor,
        vm_region_basic_info_t      info,
        mk_rpc_size_t *             infoSize,
        mach_port_t *               object_name);
```

DESCRIPTION

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

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

PARAMETERS

target_task

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

address

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

size

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

flavor

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

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

info

[out structure] Returned region information.

infoSize

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

object_name

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

NOTES

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

RETURN VALUE

KERN_INVALID_ADDRESS

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

RELATED INFORMATION

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

Data Structures: **vm_region_basic_info**

vm_reserve

Function — Marks a memory region as reserved

SYNOPSIS

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

DESCRIPTION

The **vm_reserve** function marks the specified region as reserved and is therefore protected from operations that implicitly allocate memory.

PARAMETERS

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

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

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

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

vm_unreserve

Function — Marks a memory region as no longer reserved

SYNOPSIS

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

DESCRIPTION

The **vm_unreserve** function marks the specified memory region as no longer reserved. A region marked as reserved is protected from operations that implicitly allocate memory.

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

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

RELATED INFORMATION

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

vm_unwire

Function — Resets the pageability of a region of virtual memory

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

NOTES

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

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

RETURN VALUE

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

RELATED INFORMATION

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

vm_wire

Function — Forces pages to be and stay resident

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

NOTES

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

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

RETURN VALUE

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

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

RELATED INFORMATION

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

vm_write

Function — Writes data to a task's virtual memory

SYNOPSIS

```
kern_return_t vm_write
        (mach_port_t                task,
        vm_address_t                destination_address,
        vm_address_t                source_address,
        mk_rpc_size_t               length);
```

DESCRIPTION

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

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

PARAMETERS

task

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

destination_address

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

source_address

[in scalar] The address within the calling thread's address space from which the data is to be written. The address may be anywhere within an allocated page.

length

[in scalar] The number of bytes to be written.

NOTES

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

RETURN VALUE

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

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

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

RELATED INFORMATION

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

CHAPTER 5

External Memory Management Functions

This chapter discusses the interfaces used by external memory managers to invoke paging operations on the kernel.

memory_object_change_attributes

Function — Changes various object attributes

SYNOPSIS

```
kern_return_t memory_object_change_attributes
        (mach_port_t                mem_cache_control,
        memory_object_flavor_t      flavor,
        memory_object_info_t        attributes,
        mk_rpc_size_t               attributesSize,
        mach_port_t                 notify_port);
```

DESCRIPTION

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

PARAMETERS

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

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

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

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

attributes
: [pointer to in structure] New attributes.

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

notify_port
: [in reply receive (to be converted to send) right] The notification port to which a notification is to be sent (via

memory_object_change_completed) indicating the completion of the attribute change; may be MACH_PORT_NULL.

RETURN VALUE

RELATED INFORMATION

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

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

memory_object_data_error

Function — Indicates no data for a memory object

SYNOPSIS

```
kern_return_t memory_object_data_error
        (memory_object_data_target_t    mem_cache_control,
        vm_offset_t                     offset,
        vm_size_t                       size,
        kern_return_t                   error_value);
```

DESCRIPTION

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

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

PARAMETERS

mem_cache_control

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

offset

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

size

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

error_value

[in scalar] Reason for the error.

NOTES

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

RETURN VALUE

Only RPC transmission errors apply.

RELATED INFORMATION

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

memory_object_data_supply

Function — Supplies data for a region of a memory object

SYNOPSIS

```
kern_return_t memory_object_data_supply
        (memory_object_data_target_t    mem_cache_control,
        vm_offset_t                     offset,
        vm_address_t                    data,
        vm_size_t                       length,
        vm_prot_t                       lock_value,
        boolean_t                       precious,
        mach_port_t                     reply_port);
```

DESCRIPTION

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

PARAMETERS

mem_cache_control

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

offset

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

data

[in scalar] The address of the data being provided to the kernel.

length

[in scalar] The number of bytes 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_ALL

Prohibits all forms of access.

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

VM_PROT_READ
: Prohibits read access.

VM_PROT_WRITE
: Prohibits write access.

VM_PROT_EXECUTE
: Prohibits execute access.

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

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

NOTES

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

CAUTIONS

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

RETURN VALUE

Only RPC transmission errors apply.

RELATED INFORMATION

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

memory_object_data_unavailable

Function — Indicates no data for a memory object

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

mem_cache_control

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

offset

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

size

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

RETURN VALUE

Only RPC transmission errors apply.

RELATED INFORMATION

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

memory_object_destroy

Function — Shuts down a memory object

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

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

NOTES

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

RETURN VALUE

Only RPC transmission errors apply.

RELATED INFORMATION

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

memory_object_establish

Function — Marks a memory object ready to receive paging operations

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

host_name

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

ledger

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

mem_cache_control

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

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

abs_rep
: [in abstract-memory-object receive (to be converted to send) right] The underlying abstract memory object. The kernel will transmit all subsequent notifications via this port.

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

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

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

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

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

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

RETURN VALUE

Only RPC transmission errors apply.

RELATED INFORMATION

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

Data Structures: **memory_object_behave_info**.

memory_object_get_attributes

Function — Returns current attributes for a memory object

SYNOPSIS

```
kern_return_t memory_object_get_attributes
        (mach_port_t                mem_cache_control,
        memory_object_flavor_t      flavor,
        memory_object_info_t        attributes,
        mk_rpc_size_t *             attributesSize);
```

DESCRIPTION

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

PARAMETERS

mem_cache_control

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

flavor

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

MEMORY_OBJECT_PERFORMANCE_INFO

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

MEMORY_OBJECT_BEHAVIOR_INFO

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

attributes

[out structure] Current attributes.

attributesSize

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

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

memory_object_lock_request

Function — Restricts access to memory object data

SYNOPSIS

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

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** notification using the *notify* port.

PARAMETERS

mem_cache_control

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

offset

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

size

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

should_return

[in scalar] Clean indicator. Values are:

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

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

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

should_flush

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

lock_value

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

VM_PROT_NO_CHANGE
: Do not change the protection of any pages.

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

VM_PROT_ALL
: Prohibits all forms of access.

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

VM_PROT_READ
: Prohibits read access.

VM_PROT_WRITE
: Prohibits write access.

VM_PROT_EXECUTE
: Prohibits execute access.

notify

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

NOTES

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

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

RETURN VALUE

Only RPC transmission errors apply.

RELATED INFORMATION

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

memory_object_representative

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

SYNOPSIS

```
kern_return_t memory_object_representative
        (mach_port_t                host,
        mach_port_t                 mem_cache_control,
        mach_port_t                 representative,
        mach_port_t                 abs_rep,
        vm_prot_t                   rep_max_prot,
        security_id_t               security_id);
```

DESCRIPTION

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

PARAMETERS

host

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

mem_cache_control

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

representative

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

abs_rep

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

rep_max_prot

[in scalar] The maximum protections that can be specified by any task mapping this object via the memory object representative port on which the initial **memory_object_notify** RPC was invoked.

security_id

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

NOTES

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

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

- **memory_object_representative** does not include the object attributes since the object is already initialized.
- Representative initialization may race with memory object termination. When the manager responds with **memory_object_representative**, the manager believes the memory object to be initialized. The object, though, may have been terminated but the manager has yet to receive the **memory_object_terminate** RPC. The use of **memory_object_representative** in this situation causes the kernel to ignore the RPC (since the underlying object is not known) and to reinvoke **memory_object_notify**, thereby establishing a new cache control port. When the manager handles the terminate RPC, it then re-initializes the memory object with **memory_object_establish**.

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

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

RETURN VALUE

Only RPC transmission errors apply.

RELATED INFORMATION

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

memory_object_revoke

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

mem_cache_control

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

mem_obj_rep

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

RETURN VALUE

Only RPC transmission errors apply.

RELATED INFORMATION

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

CHAPTER 6

Thread Interface

This chapter discusses the kernel's thread interfaces. This includes status functions related to threads. Properties associated with threads, such as special ports, are included here as well. Functions that apply to more than one thread appear in the task interface chapter.

mach_thread_self

System Trap — Returns the thread self port

SYNOPSIS

mach_port_t **mach_thread_self**
();

DESCRIPTION

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

PARAMETERS

None

RETURN VALUE

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

RELATED INFORMATION

Functions: **thread_set_special_port**.

thread_create

Function — Creates a thread within a task

SYNOPSIS

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

DESCRIPTION

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

The new thread holds a send right for its thread name port. A send right for the thread's name port is also returned to the calling task or thread in *child_thread*. The new thread's exception ports are set to MACH_PORT_NULL. Its scheduling policy, processor set, and PC sampling state are inherited from its parent task.

PARAMETERS

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

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_create_empty

Function — Adds a new empty thread to a port pool

SYNOPSIS

```
kern_return_t thread_create_empty
        (mach_port_t            parent_task,
        mach_port_name_t        port_pool,
        user_stack_t            service_stack,
        vm_size_t               stack_size,
        dispatch_addr_t         dispatch_function,
        vm_address_t            receive_buffer,
        vm_size_t               buffer_size,
        mach_port_t *           new_empty_thread);
```

DESCRIPTION

The **thread_create_empty** function creates a new empty thread attached to *port_pool* within *parent_task*. The new thread will use *service_stack* and *receive_buffer* to field incoming RPC invocations from ports which are members of *port_pool*. The dispatch_function parameter specifies the function to be invoked when the empty thread begins executing in user space to handle an RPC.

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

PARAMETERS

parent_task
: [in task send right] The port for the task whose port pool is to own the new thread.

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

service_stack
: [in scalar] The address of the area to be used as a stack.

stack_size
: [in scalar] The size of the area to be used as a stack.

dispatch_function
: [in scalar] The dispatch function that will be used to handle inbound RPC invocations. See **server_dispatch** for a description of this parameter.

receive_buffer
: [in scalar] The address of the area to be used as a receive buffer during RPC dispatch.

buffer_size
: [in scalar] The size of the area to be used as a receive buffer.

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

NOTES

Empty threads may be terminated explicitly via **thread_terminate**, or implicitly via destruction of the port pool to which they belong.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_create_running

Function — Optimized creation of a running thread

SYNOPSIS

```
kern_return_t thread_create_running
        (mach_port_t                parent_task,
        thread_state_flavor_t       flavor,
        thread_state_t              new_state,
        mk_rpc_size_t               new_stateSize,
        mach_port_t *               child_thread);
```

DESCRIPTION

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

The new thread is not suspended and holds a send right for its thread kernel port. A send right for the thread's name port is also returned to the calling task or thread in *child_thread*. The new thread's exception ports are set to MACH_PORT_NULL, and its scheduling policy and processor set are inherited from its parent task.

PARAMETERS

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

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

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

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

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

NOTES

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_depress_abort

Function — Cancel thread scheduling depression

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

KERN_FAILURE
The scheduling attributes of *target_thread* are not depressed.

RELATED INFORMATION

Functions: **thread_switch**.

thread_get_exception_ports

Function — Returns a send right to an exception port

SYNOPSIS

```
kern_return_t thread_get_exception_ports
        (mach_port_t                target_thread,
        exception_mask_t            exception_mask,
        exception_mask_array_t      masks,
        mk_rpc_size_t *             masksSize,
        exception_port_array_t      old_ports,
        exception_behavior_array_t  old_behaviors,
        exception_flavor_array_t    old_flavors);
```

DESCRIPTION

The **thread_get_exception_ports** function returns send rights for a specified set of exception ports belonging to *thread*. The call returns a set of quadruples <exception type mask, exception port, behavior, flavor> for each unique set of <exception port, behavior, flavor> in effect for the thread, where the exception type mask indicates for which exception types the other values apply.

PARAMETERS

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

exception_mask
: [in scalar] A flag word indicating the types of exceptions for which the exception ports are desired.

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception.

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

EXC_MASK_MACH_SYSCALL
System call with a number in the Mach call range requested.

masks
[out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

masksSize
[pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_ports
[out array of exception send rights] The returned exception ports.

old_behaviors
[out array of *exception_behavior_t*] The type of exception message to be sent. Defined types are:

EXCEPTION_STATE
Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Service interfaces: **catch_exception_raise_state**.

thread_get_special_port

Function — Returns a send right to a special port

SYNOPSIS

kern_return_t **thread_get_special_port**
(mach_port_t *target_thread,*
thread_special_port_t *which_port,*
mach_port_t * *special_port*);

thread_get_kernel_port

Macro form

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

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_get_state

Function — Returns the execution state for a thread

SYNOPSIS

```
kern_return_t thread_get_state
        (mach_port_t                target_thread,
        thread_state_flavor_t       flavor,
        thread_state_t              old_state,
        mk_rpc_size_t *             old_stateSize);
```

DESCRIPTION

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

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

PARAMETERS

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_info**, **thread_info**, **thread_set_state**.

thread_info

Function — Returns information about a thread

SYNOPSIS

```
kern_return_t thread_info
        (mach_port_t          target_thread,
        thread_flavor_t       flavor,
        thread_info_t         thread_info_out,
        mk_rpc_size_t *       thread_info_outSize);
```

DESCRIPTION

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

PARAMETERS

target_thread
: [in thread send right] The thread for which the information is to be returned.

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

THREAD_BASIC_INFO
: Returns basic information about the thread, such as the thread's run state and suspend count. The returned structure is **thread_basic_info**.

THREAD_SCHED_FIFO_INFO
: Returns FIFO scheduling policy information about the thread. The returned structure is **policy_fifo_info**

THREAD_SCHED_RR_INFO
: Returns round-robin scheduling policy information about the thread. The returned structure is **policy_rr_info**

THREAD_SCHED_TIMESHARE_INFO
: Returns timeshare scheduling policy information about the thread. The returned structure is **policy_timeshare_info**

thread_info_out
: [out structure] Information about the specified thread.

thread_info_outSize

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

NOTES

At any given time, a thread has only one scheduling policy in effect for it. Thus, only one of the scheduling information structures will be valid, that so indicated by the policy value returned by THREAD_BASIC_INFO.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

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

thread_notify

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

NOTES

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_resume

Function — Resumes a thread

SYNOPSIS

kern_return_t **thread_resume**
(mach_port_t *target_thread*);

DESCRIPTION

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

PARAMETERS

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

NOTES

An attempt to lower the suspend count below zero is ignored.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_resume**, **task_suspend**, **thread_create**, **thread_info**, **thread_suspend**, **thread_terminate**.

thread_sample

Function — Perform periodic PC sampling for a thread

SYNOPSIS

```
kern_return_t thread_sample
        (mach_port_t                    sample_thread,
        mach_port_name_t                sample_port);
```

DESCRIPTION

The **thread_sample** function causes the program counter (PC) of the specified *sample_thread* to be sampled periodically. The set of PC sample values obtained are saved in buffers that the kernel provides by invoking sample notification RPCs on the specified *sample_port*.

This function will fail to start sampling if sampling is already active on this thread due to an earlier call to either **thread_sample** or **task_sample**. This thread will fail to stop sampling if sampling was started via **task_sample**.

PARAMETERS

sample_thread
: [in thread send right] Thread whose PC is to be sampled

sample_port
: [in sample receive (to be converted to send) right] The notification port to which PC sample buffers are sent. A value of MACH_PORT_NULL stops PC sampling for the thread.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_sample**, **receive_samples, end_samples**.

thread_set_exception_ports

Function — Sets exception ports for a thread

SYNOPSIS

```
kern_return_t thread_set_exception_ports
        (mach_port_t                thread,
        exception_mask_t            exception_mask,
        exception_port_t            port,
        exception_behavior_t        behavior,
        thread_state_flavor_t       flavor);
```

DESCRIPTION

The **thread_set_exception_ports** function sets a specified set of exception ports belonging to *thread*.

PARAMETERS

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

exception_mask
: [in scalar] A flag word indicating the types of exceptions for which the exception port applies.

  EXC_MASK_BAD_ACCESS
  : Could not access memory.

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

  EXC_MASK_ARITHMETIC
  : Arithmetic exception.

  EXC_MASK_EMULATION
  : Emulation instruction. Emulation support instruction encountered.

  EXC_MASK_SOFTWARE
  : Software generated exception.

  EXC_MASK_BREAKPOINT
  : Trace, breakpoint, etc.

  EXC_MASK_SYSCALL
  : System call requested.

EXC_MASK_MACH_SYSCALL
: System call with a number in the Mach call range requested.

port
: [in exception send right] The exception port for all selected exception types.

behavior
: [in scalar] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
: Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_STATE
: Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_IDENTITY
: Send a **catch_exception_raise_state_identity** message including the thread identity and state.

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

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected; any other value causes the attempted execution of these system call numbers to be considered an exception.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_thread_self**, **task_get_exception_ports**, **task_set_exception_ports**, **task_swap_exception_ports**, **thread_create**, **thread_get_exception_ports**, **thread_swap_exception_ports**.

Service interfaces: **catch_exception_raise_state**.

thread_set_special_port

Function — Sets a special port for a thread

SYNOPSIS

kern_return_t **thread_set_special_port**
(mach_port_t *target_thread,*
thread_special_port_t *which_port,*
mach_port_t *special_port*);

thread_set_kernel_port

Macro form

kern_return_t **thread_set_kernel_port**
(mach_port_t *target_thread,*
mach_port_t *special_port)*

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

DESCRIPTION

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

PARAMETERS

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

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

THREAD_KERNEL_PORT
[thread-self port] The thread's kernel (self) port. This is the port returned by **mach_thread_self**.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_thread_self**, **task_get_special_port**, **task_set_special_port**, **thread_create**, **thread_get_special_port**.

thread_set_state

Function — Sets the execution state for a thread

SYNOPSIS

```
kern_return_t thread_set_state
        (mach_port_t                target_thread,
        thread_state_flavor_t       flavor,
        thread_state_t              new_state,
        mk_rpc_size_t               new_stateSize);
```

DESCRIPTION

The **thread_set_state** function sets the execution state (for example, the machine registers) for *target_thread*. The value of *flavor* specifies the type of state to set. The format of the state to be set is machine specific; it is defined in **<mach/thread_status.h>**.

PARAMETERS

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_suspend

Function — Suspends a thread

SYNOPSIS

kern_return_t **thread_suspend**
(mach_port_t *target_thread*);

DESCRIPTION

The **thread_suspend** function increments the suspend count for *target_thread* and prevents the thread from executing any more user-level instructions. In this context, a user-level instruction can be either a machine instruction executed in user mode or a system trap instruction, including a page fault. If a thread is currently executing within a system trap, the kernel code will continue to execute until it reaches the system return code. The system trap returns when the thread resumes.

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

PARAMETERS

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

CAUTIONS

Unpredictable results may occur if a program suspends a thread and alters its user state so that its direction is changed upon resuming.

NOTES

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

thread_suspend (*target_thread*)

thread_set_state (*target_thread*)

thread_resume (*target_thread*)

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

thread_swap_exception_ports

Function — Swaps exception ports for a thread

SYNOPSIS

```
kern_return_t thread_swap_exception_ports
        (mach_port_t                thread,
        exception_mask_t            exception_mask,
        exception_port_t            port,
        exception_behavior_t        behavior,
        thread_state_flavor_t       flavor,
        exception_mask_array_t      masks,
        mk_rpc_size_t *             masksSize,
        exception_port_array_t      old_ports,
        exception_behavior_array_t  old_behaviors,
        exception_flavor_array_t    old_flavors);
```

DESCRIPTION

The **thread_swap_exception_ports** function sets a specified set of exception ports belonging to *thread*, returning the old set.

PARAMETERS

thread

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

exception_mask

[in scalar] A flag word indicating the types of exceptions for which the exception port applies.

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

EXC_MASK_MACH_SYSCALL
System call with a number in the Mach call range requested.

port
[in exception send right] The exception port for all selected exception types.

behavior
[in scalar] Control of the behavior of the exception processing. Defined types are:

EXCEPTION_DEFAULT
Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_STATE
Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

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

masks
[out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

masksSize
[pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_ports
[out array of exception send rights] The returned exception ports.

old_behaviors
[out array of *exception_behavior_t*] The type of exception message to be sent as with *behavior.*

old_flavors

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

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected; any other value causes the attempted execution of these system call numbers to be considered an exception.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_thread_self, task_get_exception_ports, task_set_exception_ports, task_swap_exception_ports, thread_create, thread_get_exception_ports, thread_set_exception_ports.**

Service Interfaces: **catch_exception_raise_state**.

thread_switch

System Trap — Cause context switch with options

SYNOPSIS

kern_return_t **thread_switch**
(switch_option_t *option,*
mk_rpc_timeout_t *time*);

DESCRIPTION

The **thread_switch** function provides low-level access to the scheduler's context switching code. The scheduler's selection logic is invoked to select the next thread to run; the current thread may continue running if there is no other appropriate thread to run. The *option* argument specifies additional semantics.

PARAMETERS

option

[in scalar] One of the options specified.

SWITCH_OPTION_NONE

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

SWITCH_OPTION_WAIT

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

SWITCH_OPTION_DEPRESS

The thread's scheduling attributes are temporarily set so as to provide it with the lowest possible service for duration *time*. The scheduling depression is aborted when *time* has passed, when the current thread is next run (because the processor set has nothing else to run), or when **thread_depress_abort** is applied to the current thread. Changing the thread's scheduling attributes (via **thread_policy** or **thread_set_policy**) will not affect this depression. If *time* is 0, no timeout is associated with the depression

SWITCH_OPTION_IDLE

The thread's scheduling attributes are temporarily set so as to provide it with the lowest possible service for duration *time*. The scheduling depression is aborted when *time* has passed or when **thread_depress_abort** is applied to the current thread.

KERN_ABORTED is returned if the depression is cancelled via **thread_depress_abort**. Changing the thread's scheduling attributes (via **thread_policy** or **thread_set_policy**) or applying **thread_resume** will not affect this depression.

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

time

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

NOTES

thread_switch is often called when the current thread can proceed no further for some reason; the various options and arguments allow information about the thread to be transmitted to the kernel. The SWITCH_OPTION_WAIT option is used when the amount of time that the current thread must wait before it can do anything useful can be estimated and is fairly short, especially when the identity of the thread for which this thread must wait is not known. SWITCH_OPTION_IDLE effectively allows an idle system to busy wait in user space. Multiple threads with SWITCH_OPTION_IDLE depression are subject to round-robin scheduling. The time spent executing in user space under scheduling depression is recorded separately from user and system time. It can be obtained as the *user_idle_time* value from the THREAD_BASIC_INFO flavor of thread_info for a specific thread, or as the *user_idle_time* value from the TASK_BASIC_INFO or TASK_THREAD_TIMES_INFO flavors of **task_info**. SWITCH_OPTION_DEPRESS is recommended when the time cannot be reasonable estimated.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **thread_depress_abort**.

thread_terminate

Function — Destroys a thread

SYNOPSIS

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

DESCRIPTION

The **thread_terminate** function destroys *target_thread*. If *target_thread* is not the root thread for the shuttle currently attached to it, that shuttle is not terminated, but returns from the current RPC with a return value of KERN_TERMINATED.

PARAMETERS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_terminate**, **task_threads**, **thread_create**, **thread_resume**, **thread_suspend**.

CHAPTER 7

Task Interface

This chapter discusses the specifics of the kernel's task interfaces. This includes functions that return status information for a task. Also included are functions that operate upon all or a set of threads within a task.

mach_task_self

System Trap — Returns the task self port

SYNOPSIS

mach_port_t **mach_task_self**
();

DESCRIPTION

The **mach_task_self** function returns send rights to the task's kernel port.

PARAMETERS

None

NOTES

The include file **<mach_init.h>** included by **<mach.h>** redefines this function call to simply return the value **mach_task_self**, cached by the Mach run-time.

RETURN VALUE

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

RELATED INFORMATION

Functions: **task_set_special_port**.

task_create

Function — Creates a task

SYNOPSIS

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

DESCRIPTION

The **task_create** function creates a new task from *parent* and returns the name of the new task in *child*. The child task acquires shared or copied parts of the parent's address space (see **vm_inherit**). The child task initially contains no threads. The child task inherits the parent's security ID, processor set, and scheduling policy.

The child task receives the following "special" ports, which are created or copied for it at task creation:

- [task-self send right] The port by which the kernel knows the new child task and allows it to be manipulated. The child task holds a send right for this port. The port name is also returned to the calling task.
- [bootstrap send right] The port to which the child task can send a message requesting return of any system service ports that it needs (for example, a port to a Name Server). The child task inherits a send right for this port from the parent task. The task can use **task_set_special_port** to change this port.
- [host-self send right] The port by which the child task requests information about its host. The child task inherits a send right for this port from the parent task.
- [ledger send right] The port naming the ledger from which the task draws its resources.

The child task also inherits the following ports:

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

PARAMETERS

ledger

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

parent
: [in task send right] The port for the task from which to draw the child task's port rights and address space.

inherit_memory
: [in scalar] Address space inheritance indicator. If true, the child task inherits the (inheritable) address space of the parent task. If false, the kernel assigns the child task an empty address space.

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

RETURN VALUE

KERN_INVALID_TASK
: *parent* does not name a task.

RELATED INFORMATION

Functions: **task_create_security_id**, **task_resume**, **task_set_special_port**, **task_suspend**, **task_terminate**, **task_threads**, **thread_create**, **thread_resume**, **vm_inherit**, **task_sample**, **task_set_exception_ports**.

task_get_exception_ports

Function — Returns send rights to exception ports

SYNOPSIS

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

DESCRIPTION

The **task_get_exception_ports** function returns send rights for a specified set of exception ports belonging to *target_task*. A task exception port is used when a thread specific exception port returns a non-success reply. The call returns a set of quadruples <exception type mask, exception port, behavior, flavor> for each unique set of <exception port, behavior, flavor> in effect for the task where the exception type mask indicates for which exception types the other values apply.

PARAMETERS

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

exception_mask
: [in scalar] A flag word indicating the types of exceptions for which the exception ports are desired.

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
Software generated exception.

EXC_MASK_BREAKPOINT
Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

EXC_MASK_MACH_SYSCALL
System call with a number in the Mach call range requested.

EXC_MASK_NOTIFY
Notify (**thread_notify**) exception.

masks
[out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

masksCnt
[pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_ports
[out array of exception send rights] The returned exception ports.

old_behaviors
[out array of *exception_behavior_t*] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_STATE
Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self**, **thread_get_exception_ports**, **task_set_exception_ports**, **task_swap_exception_ports**, **thread_create**, **thread_set_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**.

task_get_special_port

Function — Returns a send right to a special port

SYNOPSIS

kern_return_t **task_get_special_port**
(mach_port_t *target_task,*
task_special_port_t *which_port,*
mach_port_t* *special_port*);

task_get_bootstrap_port

Macro form

kern_return_t **task_get_bootstrap_port**
(mach_port_t *target_task,*
mach_port_t* *special_port)*

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

task_get_kernel_port

Macro form

kern_return_t **task_get_kernel_port**
(mach_port_t *target_task,*
mach_port_t* *special_port)*

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

task_get_host_name_port

Macro form

kern_return_t **task_get_host_name_port**
(mach_port_t *target_task,*
mach_port_t* *special_port)*

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

task_get_ledger_port

Macro form

kern_return_t **task_get_ledger_port**
(mach_port_t *target_task,*
mach_port_t* *special_port)*

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

DESCRIPTION

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

If one task has a send right for the kernel port of another task, it can use the port to perform kernel operations for the other task. Send rights for a kernel port nor-

mally are held only by the task to which the port belongs, or by the task's parent task. Using the **mk_rpc** function, however, any task can pass a send right for its kernel port to another task.

PARAMETERS

target_task
: [in task send right] The port for the task for which to return the port's send right.

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

TASK_KERNEL_PORT
: [task-self send right] The port used to control this task. Used to send messages that affect the task. This is the port returned by **mach_task_self**.

TASK_BOOTSTRAP_PORT
: [bootstrap send right] The task's bootstrap port. Used to send messages requesting return of other system service ports.

TASK_HOST_NAME_PORT
: [host-self send right] The port used to request information of the containing host. This is the port returned by **mach_host_self**.

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

special_port
: [out task-special send right] The returned value for the port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self**, **task_create**, **task_set_special_port**, **thread_get_special_port**, **thread_set_special_port**, **mach_host_self**.

task_info

Function — Returns information about a task

SYNOPSIS

```
kern_return_t task_info
        (mach_port_t        target_task,
        task_flavor_t       flavor,
        task_info_t         task_info_out,
        mk_rpc_size_t*      task_info_outCnt);
```

DESCRIPTION

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

PARAMETERS

target_task

[in task send right] The port for the task for which the information is to be returned.

flavor

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

TASK_BASIC_INFO

Returns basic information about the task, such as the task's suspend count and number of resident pages. The structure returned is **task_basic_info**.

TASK_THREAD_TIMES_INFO

Returns system and user space run-times for live threads. The structure returned is **task_thread_times_info**.

TASK_SCHED_FIFO_INFO

Returns default FIFO scheduling policy attributes to be assigned to new threads. The structure returned is **policy_fifo_base**.

TASK_SCHED_RR_INFO

Returns default round-robin scheduling policy attributes to be assigned to new threads. The structure returned is **policy_rr_base**.

TASK_SCHED_TIMESHARE_INFO
: Returns default timeshare scheduling policy attributes to be assigned to new threads. The structure returned is **policy_timeshare_base**.

TASK_SECURITY_ID
: Returns the security ID for the task. The value returned is of type *security_id_t*

task_info_out
: [out structure] Information about the specified task.

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

NOTES

At any given time, a task has one default scheduling policy assigned to it (as returned by TASK_BASIC_INFO). As such, only one of the scheduling flavors will return valid information.

RETURN VALUE

KERN_INVALID_POLICY
: A request was made for the default scheduling policy attributes for the task but the requested policy is not the task's default policy.

RELATED INFORMATION

Functions: **task_get_special_port**, **task_set_special_port**, **task_threads**, **thread_info**, **thread_get_state**, **thread_set_state**.

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

task_resume

Function — Resume a task

SYNOPSIS

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

DESCRIPTION

The **task_resume** function decrements the suspend count for *target_task*. If the task's suspend count goes to zero, the function resumes any suspended threads within the task. To resume a given thread, the thread's own suspend count must also be zero.

PARAMETERS

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

NOTES

An attempt to lower the suspend count below zero is ignored.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_info**, **task_suspend**, **task_terminate**, **thread_info**, **thread_resume**, **thread_suspend**.

task_sample

Function — Perform periodic PC sampling for a task

SYNOPSIS

```
kern_return_t task_sample
        (mach_port_t        sample_task,
        mach_port_t         reply_port);
```

DESCRIPTION

The **task_sample** function causes the program counter (PC) of the specified *sample_task* (actually, all of the threads within *sample_task*) to be sampled periodically. The set of PC sample values obtained are saved in buffers which are sent to the specified *reply_port* in **receive_samples** messages.

PC sampling for any threads in the task set up via **thread_sample** is cancelled by this call, as if a call to **thread_sample** had been made with MACH_PORT_NULL.

PARAMETERS

sample_task
: [in task send right] Port for the task whose threads' PC are to be sampled

reply_port
: [in sample receive (to be converted to send) right] Port to which PC sample buffers are sent. A value of MACH_PORT_NULL stops PC sampling for the task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **end_samples**, **thread_sample**, **receive_samples**.

task_set_exception_ports

Function — Sets exception ports for a task

SYNOPSIS

```
kern_return_t task_set_exception_ports
        (mach_port_t            target_task,
        exception_mask_t        exception_mask,
        mach_port_t             port,
        exception_behavior_t    behavior,
        thread_state_flavor_t   flavor);
```

DESCRIPTION

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

PARAMETERS

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

exception_mask
: [in scalar] A flag word indicating the types of exceptions for which the exception port applies.

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
: Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

EXC_MASK_MACH_SYSCALL
System call with a number in the Mach call range requested.

EXC_MASK_NOTIFY
Notify (**thread_notify**) exception.

port
[in exception send right] The exception port for all selected exception types.

behavior
[in scalar] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_STATE
Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

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

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected; any other value causes the attempted execution of these system call numbers to be considered an exception.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self**, **task_get_exception_ports**, **task_swap_exception_ports**, **thread_set_exception_ports**, **thread_create**, **thread_get_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**.

task_set_special_port

Function — Sets a special port for a task

SYNOPSIS

kern_return_t **task_set_special_port**
(mach_port_t *target_task,*
task_special_port_t *which_port,*
mach_port_t *special_port*);

task_set_bootstrap_port

Macro form

kern_return_t **task_set_bootstrap_port**
(mach_port_t *target_task,*
mach_port_t *special_port)*

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

task_set_kernel_port

Macro form

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

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

task_set_host_name_port

Macro form

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

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

task_set_ledger_port

Macro form

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

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

DESCRIPTION

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

PARAMETERS

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

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

TASK_BOOTSTRAP_PORT
[bootstrap send right] The task's bootstrap port. Used to send messages requesting return of other system service ports.

TASK_KERNEL_PORT
[task-self send right] The task's kernel port. Used by the kernel to receive messages to manipulate the task. This is the port returned by **mach_task_self**. Setting this special port does not change the identity of the kernel port that names the task; this simply changes the value returned as the kernel special port.

TASK_HOST_NAME_PORT
[host-self send right] The task's host self port. Used by the task to request information about its containing host. This is the port returned by **mach_host_self**. Setting this special port does not change the identity of the kernel port that names the host; this simply changes the value returned as the host special port.

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

special_port
[in task-special send right] The value for the port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_get_special_port**, **mach_task_self**, **thread_get_special_port**, **thread_set_special_port**, **mach_host_self**.

task_suspend

Function — Suspends a task

SYNOPSIS

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

DESCRIPTION

The **task_suspend** function increments the suspend count for *target_task* and stops all threads within the task. As long as the suspend count is positive, no newly-created threads can execute. The function does not return until all of the task's threads have been suspended.

PARAMETERS

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

NOTES

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_info**, **task_resume**, **task_terminate**, **thread_suspend**.

task_swap_exception_ports

Function — Swaps exception ports for a task

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

exception_mask
: [in scalar] A flag word indicating the types of exceptions for which the exception port applies.

EXC_MASK_BAD_ACCESS
: Could not access memory.

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

EXC_MASK_ARITHMETIC
: Arithmetic exception

EXC_MASK_EMULATION
: Emulation instruction. Emulation support instruction encountered.

EXC_MASK_SOFTWARE
: Software generated exception.

EXC_MASK_BREAKPOINT
Trace, breakpoint, etc.

EXC_MASK_SYSCALL
System call requested.

EXC_MASK_MACH_SYSCALL
System call with a number in the Mach call range requested.

EXC_MASK_NOTIFY
Notify (**thread_notify**) exception.

port
[in exception send right] The exception port for all selected exception types.

behavior
[in scalar] The type of exception message to be sent. Defined types are:

EXCEPTION_DEFAULT
Send a **catch_exception_raise** message including the thread identity.

EXCEPTION_STATE
Send a **catch_exception_raise_state** message including the thread state.

EXCEPTION_STATE_IDENTITY
Send a **catch_exception_raise_state_identity** message including the thread identity and state.

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

masks
[out array of *exception_mask_t*] An array, each element being a mask specifying for which exception types the corresponding element of the other arrays apply.

masksCnt
[pointer to in/out scalar] On input, the maximum size of the array buffers; on output, the number of returned <exception type mask, exception port, behavior, flavor> sets returned.

old_ports
[out array of exception send rights] The returned exception ports.

old_behaviors
: [out array of *exception_behavior_t*] The type of exception message to be sent as with *behavior.*

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

NOTES

If the value of the EXC_MACH_SYSCALL exception class exception port is the host name port, Mach kernel traps are executed by the kernel as expected; any other value causes the attempted execution of these system call numbers to be considered an exception.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self**, **task_get_exception_ports**, **task_set_exception_ports**, **thread_create**, **thread_get_exception_ports**, **thread_set_exception_ports**, **thread_swap_exception_ports**, **catch_exception_raise**.

task_terminate

Function — Destroys a task

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

target_task
[in task send right] The task to be destroyed.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_suspend**, **task_resume**, **thread_terminate**, **thread_suspend**.

task_threads

Function — Returns a list of the threads within a task

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

target_task

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

threads

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

threadsCnt

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

CHAPTER 8

Host, Clock and Alarm Interface

This chapter discusses the specifics of the kernel's host interfaces. Included are functions that return status information for a host, such as kernel parameters.

Note that hosts are named both by a name port, which allows the holder to request information about the host, and a control port, which provides full control access. The control port for a host is provided to the bootstrap task for that host. Only the name port operations are discussed in this chapter.

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

alarm_create

Function — Create an alarm object

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

NOTES

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

RETURN VALUE

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

KERN_FAILURE
: The specified clock does not provide this service.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **alarm_set**, **alarm_cancel**, **alarm_terminate**.

alarm_cancel

Function — Cancel a pending or repeating alarm

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

alarm_set

Function — Arm an alarm object

SYNOPSIS

```
kern_return_t alarm_set
        (mach_port_t        alarm,
        alarm_type_t        alarm_type,
        tvalspec_t          alarm_time,
        tvalspec_t          repeat_time);
```

DESCRIPTION

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

PARAMETERS

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

alarm_type
: [in scalar] How to interpret the *alarm_time* value:

  TIME_RELATIVE
  : Interpret the alarm time as relative to the current time.

  TIME_ABSOLUTE
  : Interpret the alarm time as an absolute time.

alarm_time
: [in structure] The time when the first alarm is to be sent.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Data Structures: **tvalspec**.

alarm_terminate

Function — Destroy an alarm object

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

clock_get_attributes

Function — Returns attributes of a clock

SYNOPSIS

```
kern_return_t clock_get_attributes
        (mach_port_t          clock,
        clock_flavor_t        flavor,
        clock_attr_t          clock_attr,
        mk_rpc_size_t*        clock_attrCnt);
```

DESCRIPTION

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

PARAMETERS

clock

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

flavor

[in scalar] Type of information desired. Defined values are:

CLOCK_GET_TIME_RES
: The resolution, in nanoseconds, with which the value returned by **clock_get_time** is updated.

CLOCK_MAP_TIME_RES
: The resolution, in nanoseconds, with which the value visible via **clock_map_time** is updated.

CLOCK_ALARM_CURRES
: The resolution, in nanoseconds, at which clock alarm and sleep timers are currently serviced.

CLOCK_ALARM_MINRES
: The minimum resolution, in nanoseconds, at which clock alarm and sleep timers can be serviced.

CLOCK_ALARM_MAXRES
: The maximum resolution, in nanoseconds, at which clock alarm and sleep timers can be serviced.

clock_attr

[out scalar] The returned attribute.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_time**, **clock_map_time**, **clock_sleep**, **alarm_set**, **clock_set_attributes**.

clock_get_time

Function — Returns the current time

SYNOPSIS

kern_return_t **clock_get_time**
(mach_port_t *clock,*
tvalspec_t* *cur_time*);

DESCRIPTION

The **clock_get_time** function returns the current time kept by a clock. The value returned is a monotonically increasing value (unless tampered with via the **clock_set_time** function).

PARAMETERS

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

cur_time
[out structure] Current time

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_map_time**, **clock_sleep**, **alarm_set**, **clock_set_time, get_time_stamp**.

Data Structures: **tvalspec**.

clock_map_time

Function — Returns a memory object that maps a clock

SYNOPSIS

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

DESCRIPTION

The **clock_map_time** function returns a memory object representative port representing read access to a memory object that contains (at offset zero) a mapped version of the clock time. The returned right is suitable as an argument for **vm_map**.

PARAMETERS

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

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

NOTES

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

RETURN VALUE

KERN_FAILURE
The specified clock does not provide this service.

RELATED INFORMATION

Functions: **host_get_clock_service**, **clock_get_attributes**, **clock_get_time**, **clock_sleep**, **alarm_create**.

Data Structures: **mapped_tvalspec**.

clock_sleep

System Trap — Sleep until a given time

SYNOPSIS

```
kern_return_t clock_sleep
        (mach_port_t        clock_name,
        sleep_type_t        sleep_type,
        tvalspec_t          sleep_time,
        tvalspec_t*         wake_time);
```

DESCRIPTION

The **clock_sleep** system trap delays the invoking thread until a specified time.

PARAMETERS

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

sleep_type
: [in scalar] How to interpret the *sleep_time* value:

  TIME_RELATIVE
  : Interpret the sleep time as relative to the current time.

  TIME_ABSOLUTE
  : Interpret the sleep time as an absolute time.

sleep_time
: [in structure] The time when the sleep is to terminate.

wake_time
: [out structure] The actual (absolute) time at which the sleep terminated.

NOTES

If the specified time is in the past, the call returns immediately with the wake time being the current time.

If the clock's time is changed (**clock_set_time**), the sleep will be interrupted.

The thread will waken at the service time nearest the specified sleep time as governed by the current clock alarm resolution.

Not all clocks provide this service but the REALTIME clock must.

RETURN VALUE

KERN_FAILURE
: The clock does not support a sleep service.

KERN_ABORTED
: The sleep was terminated via use of **clock_set_time**.

RELATED INFORMATION

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

Data Structures: **tvalspec**.

get_time_stamp

System Trap— Quickly return high accuracy time

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

cur_time
[out structure] Current time

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **clock_get_time**.

Data Structures: **tvalspec**.

host_get_clock_service

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

SYNOPSIS

```
kern_return_t host_get_clock_service
        (mach_port_t        host,
        clock_id_t          clock_id,
        mach_port_t*        clock_name);
```

DESCRIPTION

The **host_get_clock_service** function returns a send right to the name port for a kernel clock object. This right is used to get the time and resolutions of the clock and to set clock alarms.

PARAMETERS

host

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

clock_id

[in scalar] The identification of the desired kernel clock. These values are defined in **<mach/clock_types.h>**. Although an implementation may define additional values, the following values are always defined (although only the REALTIME clock is required to be implemented):

REALTIME_CLOCK
: A moderate resolution clock service that (typically) tracks time since the system last boot.

BATTERY_CLOCK
: A (typically) low resolution clock (to the second) that survives power failures or service outages.

HIGHRES_CLOCK
: A high resolution clock.

clock_name

[out clock-name send right] Name port for the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **clock_get_time**, **clock_get_attributes**, **clock_map_time**, **clock_sleep**, **alarm_create**, **host_get_clock_control**.

host_info

Function — Returns information about a host

SYNOPSIS

```
kern_return_t host_info
        (mach_port_t        host,
        host_flavor_t       flavor,
        host_info_t         host_info_out,
        mk_rpc_size_t*      host_info_outCnt);
```

DESCRIPTION

The **host_info** function returns selected information about a host, as specified by *flavor.*

PARAMETERS

host

[in host-name send right] The name (or control) port for the host for which information is to be obtained. The name port is the same as that returned by **mach_host_self**.

flavor

[in scalar] The type of statistics desired.

HOST_BASIC_INFO

Basic information (number of processors, amount of memory). The returned structure is **host_basic_info**.

HOST_SCHED_INFO

Basic restrictions of the kernel's scheduling, minimum quantum and time-out value. The returned structure is **host_sched_info**

host_info_out

[out structure] Information about the specified host.

host_info_outCnt

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

NOTES

This interface is machine word length specific because of the memory size returned by HOST_BASIC_INFO.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_boot_info**, **host_kernel_version**, **host_statistics**, **mach_host_self**.

Data Structures: **host_basic_info, host_sched_info**.

host_kernel_version

Function — Returns kernel version information for a host

SYNOPSIS

```
kern_return_t host_kernel_version
        (mach_port_t                    host,
        kernel_version_t                version);
```

DESCRIPTION

The **host_kernel_version** function returns the version string compiled into the kernel executing on *host* at the time it was built. This describes the version of the kernel. The constant KERNEL_VERSION_MAX (in **mach/host_info.h**) should be used to dimension storage for the returned string if the *kernel_version_t* declaration is not used.

PARAMETERS

host

[in host-name send right] The name (or control) port for the host for which information is to be obtained. The name port is the same as that returned by **mach_host_self**.

version

[out array of *char*] Character string describing the kernel version executing on *host*

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**, **mach_host_self**.

mach_host_self

System Trap — Returns the host self port

SYNOPSIS

mach_port_t **mach_host_self**
();

DESCRIPTION

The **mach_host_self** function returns send rights to the task's host self port. By default, this is the name port for the current host but can be a different value if so set.

PARAMETERS

None

RETURN VALUE

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

RELATED INFORMATION

Functions: **host_info**, **task_set_special_port**.

CHAPTER 9

Kernel Device Interface

This chapter discusses the specifics of the device interfaces to in-kernel device drivers. These interfaces provide read, write and status interfaces to devices.

device_close

Function — Terminate a connection to a device.

SYNOPSIS

kern_return_t **device_close**
(mach_port_t *device*);

DESCRIPTION

The **device_close** function destroys the associated device port. The open count for the named device is decremented. If this count reaches zero, the close operation of the device driver is invoked, closing the device.

PARAMETERS

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

NOTES

device_close will destroy any mapped device windows and other indirect device references obtained through this device port.

RETURN VALUE

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

RELATED INFORMATION

Functions: **device_open**, **memory_object_revoke**.

device_get_status

Function — Return the current device status

SYNOPSIS

```
kern_return_t device_get_status
        (mach_port_t            device,
        dev_flavor_t            flavor,
        dev_status_t            status,
        mk_rpc_size_t *         statusSize);
```

DESCRIPTION

The **device_get_status** function returns status information pertaining to an open device. The possible values for *flavor* as well as the meaning of the returned status information is device dependent.

PARAMETERS

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

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

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

statusSize
: [pointer to in/out scalar] On input, the reserved size of *status*; on output, the size of the returned device status (in natural-sized units).

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_INVALID_OPERATION
: The *flavor* requested is inconsistent with the device type or opening mode.

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

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

RELATED INFORMATION

Functions: **device_set_status**.

device_map

Function — Establish a memory manager representing a device

SYNOPSIS

```
kern_return_t device_map
        (mach_port_t        device,
        vm_prot_t           prot,
        vm_size_t*          size,
        mach_port_t*        pager);
```

DESCRIPTION

The **device_map** function establishes a memory manager that presents a memory object representing a device. The resulting port is suitable for use as the memory manager port in a **vm_map** call. This call is device dependent.

PARAMETERS

device

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

prot

[in scalar] Protection for the device memory.

size

[out scalar] The size of the device memory object. All mappings of the same device receive the same size.

pager

[out memory-object-representative send right] The returned memory object representative port to a memory manager that represents the device.

NOTES

Port rights are maintained as follows:

Memory object representative port (and abstract memory object port) — The device memory manager has all rights.

Memory cache control port — The device memory manager has only send rights.

Regardless how the object is created, the control port is created by the kernel and passed through the memory management interface.

If the device port used is restricted to use by a single identity, the generated representative port will be likewise restricted.

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

CAUTIONS

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

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

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

D_READ_ONLY
: Data cannot be written to this device.

RELATED INFORMATION

Functions: **vm_map**.

device_packet_request

Function — Request an asynchronous read operation from a network device.

SYNOPSIS

```
kern_return_t device_packet_request
        (mach_port_t        device,
        mach_port_t         completion_port,
        unsigned int        request_id,
        pointer_t           data,
        mk_rpc_size_t       dataSize);
```

DESCRIPTION

The **device_packet_request** function requests that the kernel read a data packet from a specific network device object to a region in the caller's virtual address space. The caller must explicitly allocate this region before invoking **device_packet_request**.

The kernel invokes **device_packet_complete** on a notification port specified by the caller to report the operation's completion.

Before packets can be read from a particular device, a packet selection filter must be installed on behalf of the device with a call to the **device_set_filter** interface.

PARAMETERS

device

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

completion_port

[in notification port receive right] The port that shall be used by the kernel to indicate completion of the requested operation.

request_id

[in scalar] An item that can be used to uniquely identify the current request upon the operation's completion. The contents and meaning of this parameter is entirely specific to the caller; the parameter's value is wholly opaque to the kernel.

data

[in pointer to preallocated read buffer] Returned data bytes.

dataSize
: [in scalar] The requested number bytes to read.

RETURN VALUE

D_INVALID_SIZE
: Invalid I/O size. The size must be positive.

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

D_INVALID_OPERATION
: The target device is not a network device, or the device's packet selection filter is not set.

RELATED INFORMATION

Functions: **device_read_packet**, **device_set_filter**, **device_packet_complete**.

device_read

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

SYNOPSIS

```
kern_return_t device_read
        (mach_port_t                device,
        dev_mode_t                  mode,
        recnum_t                    recnum,
        pointer_t                   data,
        mk_rpc_size_t *             dataSize);
```

DESCRIPTION

The **device_read** function reads a sequence of bytes from a device object to a region in the caller's virtual address space. The caller must explicitly allocate this region before invoking **device_read**. The characteristics of the actual read operation is device-dependent; the meaning of the *recnum* parameter is also device-dependent.

PARAMETERS

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

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read; used for positioning of random access devices.

data
: [in pointer to preallocated read buffer] Returned data bytes.

dataSize
: [in/out scalar] On input, requested number bytes to read. On output, the number of bytes actually read.

RETURN VALUE

In addition to the error codes listed below, **device_read** may return a device-specific error code if that error caused the operation to fail completely:

D_DEVICE_DOWN
: Device has been shut down.

D_INVALID_RECNUM
: Invalid record (block) number.

D_INVALID_SIZE
: Invalid I/O size.

D_IO_ERROR
: Hardware I/O error.

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

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

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set.

RELATED INFORMATION

Functions: **device_read_request**, **device_write, device_write_request**.

device_read_packet

Function — Read a data packet from a network device object.

SYNOPSIS

```
kern_return_t device_read_packet
        (mach_port_t          device,
         dev_mode_t           mode,
         net_header_t         header,
         pointer_t            data,
         mk_rpc_size_t *      dataSize);
```

DESCRIPTION

The **device_read_packet** function reads a data packet from a network device object to a region in the caller's virtual address space. The caller must explicitly allocate this region before invoking **device_read_packet**.

Before packets can be read from a particular device, a packet selection filter must be installed on behalf of the device with a call to the **device_set_filter** interface.

PARAMETERS

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

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

header
: [out array of bytes] When the call returns successfully, this parameter contains the packet's network header.

data
: [in pointer to preallocated read buffer] The area in which the packet is to be read.

dataSize
: [in/out scalar] On input, this parameter specifies the maximum packet size the caller is prepared to read; on output, this parameter specifies the size of the packet actually read.

RETURN VALUE

In addition to the errors described below, **device_read_packet** may return a device-specific error code if that error caused the operation to fail completely:

D_INVALID_SIZE
: Invalid I/O size.

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

D_INVALID_OPERATION
: The target device is not a network device, or the device's packet selection filter is not set.

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set.

RELATED INFORMATION

Functions: **device_packet_request**, **device_set_filter**.

device_read_request

Function — Request an asynchronous read operation on a specific device.

SYNOPSIS

```
kern_return_t device_read_request
        (mach_port_t            device,
        mach_port_t             completion_port,
        unsigned int            request_id,
        dev_mode_t              mode,
        recnum_t                recnum,
        pointer_t               data,
        mk_rpc_size_t           dataSize);
```

DESCRIPTION

The **device_read_request** function requests that a sequence of bytes from a specific device object be read to a region in the caller's virtual address space. The caller must explicitly allocate this region before invoking **device_read_request**. The characteristics of the actual read operation is device-dependent; the meaning of the *recnum* parameter is also device-dependent.

The kernel invokes **device_read_complete** on a notification port specified by the caller to report the operation's completion.

PARAMETERS

device

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

completion_port

[in notification port receive right] The port that shall be used by the kernel to indicate completion of the requested operation.

request_id

[in scalar] An item that can be used to uniquely identify the current request upon the operation's completion. The contents and meaning of this parameter is entirely specific to the caller; the parameter's value is wholly opaque to the kernel.

mode

[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT

Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read; used for positioning of random access devices.

data
: [in pointer to preallocated read buffer] Returned data bytes.

dataSize
: [in scalar] The requested number bytes to read.

RETURN VALUE

RELATED INFORMATION

Functions: **device_read**, **device_read_complete**, **device_write**, **device_write_request**.

device_set_filter

Function — Associate a packet selection filter for an open device instance.

SYNOPSIS

```
kern_return_t device_set_filter
        (mach_port_t                 device,
        net_priority_t               priority,
        const filter_array_t         filter,
        mk_rpc_size_t                filterCnt);
```

DESCRIPTION

It is typically the case that a packet-based network device supplies data to multiple kernel device objects. Each of these objects represents an open instance of the underlying network device and will be referred to below as *open devices*. The **device_set_filter** function allows the programmer to associate a packet selection filter with a specific open device so that packets that should be received via the open device can be identified and routed appropriately.

Each open device may be associated with one and only one filter at a time; an open device's current filter can be disabled or otherwise modified by another call to **device_set_filter**.

When a packet appears on the network device, the interrupt handler arranges for all open device filters to process the packet. The **device_set_filter** function allows the programmer to specify a filter's priority relative to other open device filters. Filters whose priorities are relatively high will see incoming packets before filters whose priorities are relatively low, but note that all filters will see all packets.

Each filter is implemented via an array of commands; each entry in the array contains a 16 bit value that constitutes a single command. The maximum number of commands per filter is specified by NET_MAX_FILTER.

A filter's command array specifies a sequence of actions that leave a boolean value on the top of an internal stack. Each 16-bit value of the command list specifies a data (push) operation (high order NETF_NBPO bits) as well as a binary operator (low order NETF_NBPA bits).

Filter processing terminates in either of two cases: the filter command list is emptied, or a NETF_C... operation terminates the list. If the final value of the top of the stack is TRUE, the message is accepted for the filter.

The value to be pushed onto the stack is chosen as follows.

NETF_PUSHLIT
: Use the next 16-bit value of the filter as the value.

NETF_PUSHZERO
: Use 0 as the value.

NETF_PUSHWORD+*N*
: Use 16-bit value *N* of the “data” portion of the message as the value.

NETF_PUSHHDR+*N*
: Use 16-bit value *N* of the “header” portion of the message as the value.

NETF_PUSHIND
: Pops the top 32-bit value from the stack and then uses it to index the 16-bit value of the “data” portion of the message to be used as the value.

NETF_PUSHHDRIND
: Pops the top 32-bit value from the stack and then uses it to index the 16-bit value of the “header” portion of the message to be used as the value.

NETF_PUSHSTK+*N*
: Use 32-bit value *N* of the stack (where the top of stack is value 0) as the value.

NETF_NOPUSH
: Don’t push a value.

The unsigned value so chosen is promoted to a 32-bit value before being pushed.

Once a value is pushed (except for the case of NETF_NOPUSH), the top two 32-bit values of the stack are popped and a binary operator applied to them (with the old top of stack as the second operand). The result of the operator is pushed on the stack. These operators are:

NETF_NOP
: Don’t pop off any values and do no operation.

NETF_EQ
: Perform an equal comparison.

NETF_LT
: Perform a less than comparison.

NETF_LE
: Perform a less than or equal comparison.

NETF_GT
: Perform a greater than comparison.

NETF_GE
: Perform a greater than or equal comparison.

NETF_AND
: Perform a bit-wise boolean AND operation.

NETF_OR
: Perform a bit-wise boolean inclusive OR operation.

NETF_XOR
: Perform a bit-wise boolean exclusive OR operation.

NETF_NEQ
: Perform a not equal comparison.

NETF_LSH
: Perform a left shift operation.

NETF_RSH
: Perform a right shift operation.

NETF_ADD
: Perform an addition.

NETF_SUB
: Perform a subtraction.

NETF_COR
: Perform an equal comparison. If the comparison is TRUE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CAND
: Perform an equal comparison. If the comparison is FALSE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CNOR
: Perform a not equal comparison. If the comparison is FALSE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

NETF_CNAND
: Perform a not equal comparison. If the comparison is TRUE, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

PARAMETERS

device
: [in device send right] A port representing the target open device instance.

priority
: [in scalar] Specifies the filter's priority relative to filters associated with other open instances of the underlying network device.

filter
: [pointer to in array of *filter_t*] The filter's command list.

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

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_INVALID_OPERATION
: The target is not a network device.

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

RELATED INFORMATION

Functions: **device_read_packet**, **device_packet_request**.

device_set_status

Function — Sets device status.

SYNOPSIS

```
kern_return_t device_set_status
        (mach_port_t        device,
        dev_flavor_t        flavor,
        dev_status_t        status,
        mk_rpc_size_t       statusSize);
```

DESCRIPTION

The **device_set_status** function sets the specified device's status. The possible values of *flavor* as well as the corresponding meanings are specific to the device.

PARAMETERS

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

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

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

statusSize
: [in scalar] The size of the status information (in natural-sized units).

RETURN VALUE

D_DEVICE_DOWN
: Device has been shut down

D_INVALID_OPERATION
: The flavor requested is inconsistent with the device type or opening mode.

D_IO_ERROR
: Hardware IO error

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

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

D_READ_ONLY
: Data cannot be written to this device.

RELATED INFORMATION

Functions: **device_get_status**.

device_write

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

SYNOPSIS

```
kern_return_t device_write
        (mach_port_t            device,
        dev_mode_t              mode,
        recnum_t                recnum,
        pointer_t               data,
        mk_rpc_size_t *         dataSize);
```

DESCRIPTION

The **device_write** function writes a sequence of bytes to a specified device object. The characteristics of the actual write operation is device-dependent; the meaning of the *recnum* parameter is also device-dependent.

PARAMETERS

device

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

mode

[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT

Do not wait for I/O completion.

recnum

[in scalar] Device-dependent record number that specifies where the data is to be written. This parameter is applicable to operations performed on random access devices.

data

[pointer to in array of bytes] Data to be written.

dataSize

[in/out scalar] On input, requested number bytes to write. On output, the number of bytes actually written.

RETURN VALUE

The return value from **device_write** can be a device error code if that error caused the operation to fail completely.

D_DEVICE_DOWN
: Device has been shut down.

D_INVALID_RECNUM
: Invalid record (block) number.

D_INVALID_SIZE
: Invalid IO size.

D_IO_ERROR
: Hardware IO error.

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

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

D_READ_ONLY
: Data cannot be written to this device.

D_WOULD_BLOCK
: Operation would block, but D_NOWAIT set

RELATED INFORMATION

Functions: **device_read**, **device_read_request**, **device_write_request**.

device_write_request

Function — Request that a sequence of bytes be written to a specific device.

SYNOPSIS

```
kern_return_t device_write_request
        (mach_port_t            device,
         mach_port_t            completion_port,
         unsigned int           request_id,
         dev_mode_t             mode,
         recnum_t               recnum,
         pointer_t              data);
         mk_rpc_size_t          dataSize);
```

DESCRIPTION

The **device_write_request** function requests the writing of a sequence of bytes from the caller's virtual address space to the target device. The characteristics of the actual write operation is device-dependent; the meaning of the *recnum* parameter is also device-dependent.

The kernel invokes **device_write_complete** on the notification port specified by the caller to report the operation's completion.

PARAMETERS

device

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

completion_port

[in notification port receive right] The port that shall be used by the kernel to indicate completion of the requested operation.

request_id

An item that can be used to uniquely identify the current request upon the operation's completion. The contents and meaning of this parameter is entirely specific to the caller; the parameter's value is wholly opaque to the kernel.

mode

[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT

Do not wait for I/O completion.

recnum
: [in scalar] Record number to be written; used for positioning of random access devices.

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

dataSize
: [in scalar] The requested number bytes to write.

RETURN VALUE

RELATED INFORMATION

Functions: **device_read**, **device_read_request**, **device_write**, **device_write_completion**.

CHAPTER 10 Synchronizers

This chapter discusses real time synchronizer primitives. The real time primitives are comprised of mutexes, semaphores and conditional variables.

cv_broadcast

Function — conditional variable broadcast

SYNOPSIS

kern_return_t **cv_broadcast**
(mach_port_t *cond_variable);

DESCRIPTION

The **cv_broadcast** function schedules all threads to run which are waiting for the conditional variable to be signaled.

PARAMETERS

con_variable
[in cond-variable send right] The port for the conditional variable.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **cv_create, cv_wait, cv_signal.**

cv_create

Function — Creates a conditional variable

SYNOPSIS

```
kern_return_t cv_create
        (mach_port_t        task,
        sync_policy_t       policy,
        mach_port_t         *cond_variable);
```

DESCRIPTION

The **cv_create** function creates a conditional variable that is owned by the task.

PARAMETERS

task
: [in task send right] The owning task of the created conditional variable.

policy
: [in synchronization policy] Scheduling policy of either SYNC_POLICY_FIFO or SYNC_POLICY_FIXED_PRIORITY. This scheduling policy is applied to all threads waiting for the conditional variable to be signaled.

cond_variable
: [out cond-variable send right] The port for the new conditional variable.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self.**

cv_destroy

Function — Destroys a conditional variable

SYNOPSIS

kern_return_t **cv_destroy**
(mach_port_t *cond_variable);

DESCRIPTION

The **cv_destroy** function destroys a conditional variable. The conditional variable must be owned by the current thread's task or the call fails. All threads waiting for the conditional variable to be signaled will return with the error KERN_OBJECT_DESTROYED.

PARAMETERS

cond_variable
[in cond-variable send right] The port for the conditional variable.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self, cv_create.**

cv_signal

Function — conditional variable signal

SYNOPSIS

kern_return_t **cv_signal**
(mach_port_t *cond_variable);

DESCRIPTION

The **cv_signal** function schedules a single thread to run which is waiting for the conditional variable to be signaled.

PARAMETERS

con_variable
[in cond-variable send right] The port for the conditional variable.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **cv_create, cv_wait, cv_broadcast.**

cv_wait

Function — conditional variable wait

SYNOPSIS

kern_return_t **cv_wait**
(mach_port_t *cond_variable*,
mach_port_t *lock*);

DESCRIPTION

The **cv_wait** function causes the current thread to sleep until the conditional variable is signaled. The specified user_lock is released during the sleep and re-acquired before the interface returns.

PARAMETERS

cond_variable
[in cond-variable send right] The port for the conditional variable.

lock
[in user-lock send right] The port for the mutex.

RETURN VALUE

KERN_OBJECT_DESTROYED
The mutex was destroyed while the sleep was in progress, so was not reacquired.

RELATED INFORMATION

Functions:**cv_create, cv_signal, cv_broadcast.**

cv_wait_timed

Function — conditional variable wait with timeout

SYNOPSIS

```
kern_return_t cv_wait_timed
        (mach_port_t        cond_variable,
        mach_port_t         lock,
        tvalspec_t          sleep_time);
```

DESCRIPTION

The **cv_wait_timed** function is identical in function to cv_wait, but will wait only as long as the specified duration. The specified user_lock is released during the sleep and reacquired before the interface returns.

PARAMETERS

con_variable
: [in cond-variable send right] The port for the conditional variable.

lock
: [in user-lock send right] The port for the mutex.

sleep_time
: [in tvalspec_t] The time period to sleep. A time period of zero is equivalent to calling **cv_wait.**

RETURN VALUE

KERN_OBJECT_DESTROYED
: The mutex was destroyed while the sleep was in progress, so was not reacquired.

RELATED INFORMATION

Functions: **cv_create, cv_signal, cv_wait, cv_broadcast.**

lock_acquire

Function — Acquires mutex ownership

SYNOPSIS

kern_return_t **lock_acquire**
(mach_port_t *lock);*

DESCRIPTION

The **lock_acquire** function attempts to obtain ownership of the mutex. If the mutex is already owned, then the thread waits until ownership can be assigned to it.

PARAMETERS

lock
[in user-lock send right] The port for the mutex.

RETURN VALUE

KERN_OBJECT_DESTROYED
The mutex was destroyed while the sleep was in progress, so was not acquired.

RELATED INFORMATION

Functions: **lock_create, lock_release.**

lock_create

Function — Creates a mutex

SYNOPSIS

```
kern_return_t lock_create
        (mach_port_t        task,
        sync_policy_t       policy,
        mach_port_t         *user_lock);
```

DESCRIPTION

The **lock_create** function creates a mutex that is owned by the task.

PARAMETERS

task

[in task send right] The owning task of the created mutex.

policy

[in synchronization policy] Scheduling policy of either SYNC_POLICY_FIFO or SYNC_POLICY_FIXED_PRIORITY. This scheduling policy is applied to all threads waiting for the user lock to be released.

user_lock

[out user-lock send right] The port for the new mutex.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self.**

lock_destroy

Function — Destroys a mutex

SYNOPSIS

kern_return_t **lock_destroy**
(mach_port_t *lock);*

DESCRIPTION

The **lock_destroy** function destroys a user mutex. The mutex must be owned by the current thread's task. All threads waiting for the mutex to be released will return with the error KERN_OBJECT_DESTROYED.

PARAMETERS

lock
[in user-lock send right] The port for the mutex.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **lock_create.**

lock_release

Function — Releases a mutex

SYNOPSIS

kern_return_t **lock_release**
(mach_port_t *lock);*

DESCRIPTION

The **lock_release** function relinquishes ownership of the mutex. If any threads are sleeping on the mutex waiting for ownership, then one is awakened in order to claim ownership.

PARAMETERS

lock
[in user-lock send right] The port for the mutex.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **lock_create, lock_acquire, lock_try.**

lock_try

Function — Attempts to acquire mutex ownership

SYNOPSIS

kern_return_t **lock_try**
(mach_port_t *lock);*

DESCRIPTION

The **lock_try** function attempts to obtain ownership of the mutex. If the mutex is already owned, then the call returns KERN_LOCK_OWNED.

PARAMETERS

lock
[in user-lock send right] The port for the mutex.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **lock_create, lock_release, lock_acquire.**

semaphore_create

Function — Creates semaphore

SYNOPSIS

```
kern_return_t semaphore_create
        (mach_port_t        task,
        sync_policy_t       policy,
        int                 value,
        mach_port_t         *semaphore);
```

DESCRIPTION

The **semaphore_create** function creates a semaphore.

PARAMETERS

task

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

policy

[in synchronization policy] Scheduling policy of either SYNC_POLICY_FIFO or SYNC_POLICY_FIXED_PRIORITY. This scheduling policy is applied to all threads waiting for the semaphore to be signaled.

value

[in scalar] The semaphore's initial value.

semaphore

[out semaphore send right] The port for the new semaphore.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **mach_task_self.**

semaphore_destroy

Function — Destroys semaphore

SYNOPSIS

kern_return_t **semaphore_destroy**
(mach_port_t *semaphore);*

DESCRIPTION

The **semaphore_destroy** function destroys the semaphore. The executing thread must be owned by the task which created the semaphore. All threads waiting for the semaphore to be signaled will return with the error KERN_OBJECT_DESTROYED.

PARAMETERS

semaphore
[in semaphore send right] The port for the semaphore.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **semaphore_create.**

semaphore_read_value

Function — Returns the current semaphore count

SYNOPSIS

```
kern_return_t semaphore_read_value
        (mach_port_t        semaphore,
        int                 *count);
```

DESCRIPTION

The **semaphore_read_value** function returns the current semaphore count.

PARAMETERS

semaphore

[in semaphore send right] A semaphore port for which information is desired.

count

[out scalar] The semaphore count. Note that if other threads have access to this semaphore, this count may no longer be valid.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: semaphore_create.**

semaphore_signal

Function — Increments the semaphore count by one

SYNOPSIS

kern_return_t **semaphore_signal**
(mach_port_t *semaphore);*

DESCRIPTION

The **semaphore_signal** function increments the semaphore count by one. If the increment results in semaphore value being less than or equal to zero and a thread is waiting for semaphore availability, then the thread is scheduled to run.

PARAMETERS

semaphore
[in semaphore send right] A semaphore port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: semaphore_create and semaphore_wait**.

semaphore_wait

Function — Waits for semaphore resource availability

SYNOPSIS

kern_return_t **semaphore_wait**
(mach_port_t *semaphore);*

DESCRIPTION

The **semaphore_wait** function decrements the semaphore count. A count of less than zero results in the calling thread waiting for semaphore resource availability.

PARAMETERS

semaphore
[in semaphore send right] The port for the semaphore.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **semaphore_create, semaphore_signal.**

semaphore_wait_timed

Function — Attempts to acquire semaphore resource and wait a finite time when unavailable

SYNOPSIS

kern_return_t **semaphore_wait_timed**
(mach_port_t *semaphore,*
tvalspec_t *sleep_time);*

DESCRIPTION

The **semaphore_wait_timed** function is identical in function to **semaphore_wait**, but will wait only as long as the specified duration.

PARAMETERS

semaphore
[in semaphore send right] The port for the semaphore.

sleep_time
[in tvalspec_t] The time period to sleep. A time period of zero is equivalent to having called **semaphore_wait.**

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **semaphore_create, semaphore_signal, semaphore_wait.**

CHAPTER 11

Service Interfaces

This chapter discusses the specifics of the service interfaces called by the kernel.

alarm_notify

Server Interface — Indicates that an alarm went off

SYNOPSIS

```
kern_return_t alarm_notify
        (mach_port_t      alarm_reply_port,
        kern_return_t     alarm_code,
        alarm_type_t      alarm_type,
        tvalspec_t        wake_time);
```

DESCRIPTION

An **alarm_notify** function is called as the result of a message from the kernel indicating that a previously requested alarm time (**alarm_set**) has arrived.

PARAMETERS

alarm_reply_port
: [in alarm (receive) right] The notify port named in the corresponding **alarm_create** call.

alarm_code
: [in scalar] The reply status code from the alarm. The possible values are:

  KERN_SUCCESS
  : The alarm was delivered without problem.

  KERN_ABORTED
  : The alarm was canceled.

alarm_type
: [in scalar] The alarm type value supplied to the **alarm_set** call. Only the low order byte of this value is used by the kernel so the high order bytes are available for application use as an alarm identifier.

wake_time
: [in structure] The time when the alarm RPC was sent.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **host_get_clock_service**, **alarm_create**, **alarm_set**, **alarm_cancel**, **alarm_notify_server**, **clock_set_time**.

Data Structures: **tvalspec**.

catch_exception_raise

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

SYNOPSIS

```
kern_return_t catch_exception_raise
        (mach_port_t            exception_port,
        mach_port_t             thread,
        mach_port_t             task,
        exception_type_t        exception,
        exception_data_t        code);
```

catch_exception_raise_state

```
kern_return_t catch_exception_raise_state
        (mach_port_t            exception_port,
        exception_type_t        exception,
        exception_data_t        code,
        thread_state_flavor_t   flavor,
        thread_state_t          state,
        mk_rpc_size_t           stateCnt);
```

catch_exception_raise_state_identity

```
kern_return_t catch_exception_raise_state_identity
        (mach_port_t            exception_port,
        mach_port_t             thread,
        mach_port_t             task,
        exception_type_t        exception,
        exception_data_t        code,
        thread_state_flavor_t   flavor,
        thread_state_t          state,
        mk_rpc_size_t           stateCnt);
```

DESCRIPTION

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

The alternate message forms (the format being selected when the exception port was set) indicate the inclusion of additional thread state.

PARAMETERS

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

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

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

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

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

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

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

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

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

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

EXC_SYSCALL
: System call requested. Details in *code* field.

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

EXC_NOTIFY
Notify (**thread_notify**) exception.

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

flavor
[in scalar] The type of state included as selected when the exception port was set.

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

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

NOTES

When an exception occurs in a thread, the kernel sends an exception RPC to its exception port, blocking the thread in the kernel waiting for the RPC to complete. It is assumed that some task is listening to this port, using the **exception_server** function to decode each RPC and invoke the appropriate exception handling interface to handle the exception and decide the course of action for *thread*.

If the thread should continue from the point of exception, the handler routine simply returns. If some other action should be taken, the thread's state can be explicitly set via **thread_set_state**. Either way, the thread will not proceed from the point of exception until explicitly continued.

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

For each exception type, a thread can have two exception ports defined: its thread type-specific exception port and the task type-specific exception port. The kernel will send a single exception RPC. The RPC is sent to the thread exception port if that port is set; otherwise the message is sent to the task exception port. If neither port is set, the thread is terminated.

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **exception_server**, **task_get_exception_ports**, **thread_get_exception_ports**, **thread_get_state**, **task_set_exception_ports, thread_set_exception_ports**, **thread_set_state**, **thread_terminate.**

device_packet_complete

Server Interface — Report the completion of an asynchronous read operation.

SYNOPSIS

void **device_packet_complete**
(mach_port_t *completion_port,*
unsigned int *request_id*,
kern_return_t *return_code,*
net_header_t *header,*
io_buf_len_t *dataCnt*);

DESCRIPTION

The kernel invokes the **device_packet_complete** interface to indicate the completion of a specific asynchronous packet read operation and to return status with respect to that operation.

PARAMETERS

completion_port

[in notification port send right] The port used by the kernel to indicate the operation's completion.

request_id

[in scalar] An item that identifies the request corresponding to this completion notification. The contents and meaning of this parameter is wholly opaque to the kernel; the parameter is an exact copy of an item supplied by the user via **device_packet_request**.

return_code

[in scalar] Specifies the status of the completed operation.

header

[in array of bytes] Specifies the location of the read packet's network header data structure.

dataCnt

[in scalar] The size of the packet received during the operation.

RETURN VALUE

RELATED INFORMATION

Functions: **device_set_filter, device_packet_request**.

device_read_complete

Server Interface — Report the completion of an asynchronous read operation.

SYNOPSIS

void **device_read_complete**
(mach_port_t *completion_port,*
unsigned int *request_id*,
kern_return_t *return_code,*
io_buf_len_t *dataCnt*);

DESCRIPTION

The kernel invokes the **device_read_complete** interface to indicate the completion of a specific asynchronous packet read operation and to return status with respect to that operation.

PARAMETERS

completion_port

[in notification port send right] The port used by the kernel to indicate the operation's completion.

request_id

[in scalar] An item that identifies the request corresponding to this completion notification. The contents and meaning of this parameter is wholly opaque to the kernel; the parameter is an exact copy of an item supplied by the user via **device_read_request**.

return_code
[in scalar] Specifies the status of the completed operation.

dataCnt
[in scalar] Specifies the number of bytes read during the operation.

RETURN VALUE

RELATED INFORMATION

Functions: **device_read_request**.

device_write_complete

Server Interface — Report the completion of an asynchronous write operation.

SYNOPSIS

```
void device_write_complete
        (mach_port_t        completion_port,
        unsigned int        request_id,
        kern_return_t       return_code,
        io_buf_len_t        dataCnt);
```

DESCRIPTION

The kernel invokes the **device_write_complete** interface to indicate the completion of a specific asynchronous packet read operation and to return status with respect to that operation.

PARAMETERS

completion_port

[in notification port send right] The port used by the kernel to indicate the operation's completion.

request_id

[in scalar] An item that identifies the request corresponding to this completion notification. The contents and meaning of this parameter is wholly opaque to the kernel; the parameter is an exact copy of an item supplied by the user via **device_write_request**.

return_code

[in scalar] Specifies the status of the completed operation.

dataCnt

[in scalar] The number of bytes written during the operation.

RETURN VALUE

RELATED INFORMATION

Functions: **device_write_request**.

do_mach_notify_dead_name

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

SYNOPSIS

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

DESCRIPTION

A **do_mach_notify_dead_name** function is called by **notify_server** as the result of a kernel RPC indicating that the port name is now dead as the result of the associated receive right having died.

PARAMETERS

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

name
[in scalar] The dead name.

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **notify_server**, **mk_rpc**, **port_request_notification.**

end_samples

Server Interface — Handles the occurrence of a program counter (PC) sampling termination RPC

SYNOPSIS

```
kern_return_t end_samples
        (mach_port_t        sample_port,
        sample_count_t      samples_lost);
```

DESCRIPTION

An **end_samples** function is called by **sample_server** as the result of a kernel RPC indicating that program counter sampling has been terminated. either due to setting the sample port to MACH_PORT_NULL, or because the sampled object (thread or task) has terminated. *sample_port* is the port named via **host_sample**, **task_sample** or **thread_sample**.

PARAMETERS

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

samples_lost
: [out scalar] The number of samples that were lost because both sample buffer messages were waiting to be received from *sample_port*.

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **host_sample, task_sample**, **thread_sample**, **sample_server**.

receive_samples

Server Interface — Handles the occurrence of a program counter (PC) sampling RPC

SYNOPSIS

```
kern_return_t receive_samples
        (mach_port_t        sample_port,
        sample_count_t      samples_lost,
        sample_array_t      samples,
        mk_rpc_size_t       samplesCnt);
```

DESCRIPTION

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

PARAMETERS

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

samples_lost
: [out scalar] The number of samples that were lost because both sample buffer messages were waiting to be received from *sample_port*.

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

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

NOTES

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

RETURN VALUE

Irrelevant.

RELATED INFORMATION

Functions: **host_sample, task_sample**, **thread_sample**, **sample_server**.

CHAPTER 12

External Memory Management Service Interfaces

This chapter discusses the interfaces of an external memory manager invoked by the kernel as part of the kernel's external memory management protocols.

memory_object_change_completed

Server Interface — Indicates the handling of a previous attribute change RPC

SYNOPSIS

```
kern_return_t memory_object_change_completed
        (mach_port_t                notify_port,
        mach_port_t                 mem_cache_control,
        memory_object_flavor_t      flavor);
```

DESCRIPTION

The kernel invokes the **memory_object_change_completed** RPC to notify the memory manager that it has performed all operations corresponding the memory manager's previous invocation of the **memory_object_change_attributes** RPC.

When the kernel completes the requested changes, it invokes **memory_object_change_completed** on the notification port provided in the **memory_object_change_attributes** call.

PARAMETERS

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

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

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

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

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

NOTES

The kernel does not issue **memory_object_change_completed** notifications in response to requests that cause the termination of the target object. Instead, the kernel issues a **memory_object_terminate** notification.

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_change_attributes**

memory_object_data_request

Server Interface — Requests data from a memory object

SYNOPSIS

```
kern_return_t memory_object_data_request
        (mach_port_t        abs_mem_port,
        mach_port_t         mem_cache_control,
        vm_offset_t         offset,
        vm_size_t           length,
        vm_prot_t           desired_access);
```

DESCRIPTION

The kernel invokes the **memory_object_data_request** RPC to request access to data from the specified memory object.

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

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

PARAMETERS

abs_mem_port

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

mem_cache_control

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

offset

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

length

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

desired_access

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

VM_PROT_READ
: Allows read access.

VM_PROT_WRITE
: Allows write access.

VM_PROT_EXECUTE
: Allows execute access.

RETURN VALUE

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

RELATED INFORMATION

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

memory_object_data_return

Server Interface — Writes data back to a memory object

SYNOPSIS

```
kern_return_t memory_object_data_return
        (mach_port_t        abs_mem_port,
        mach_port_t         mem_cache_control,
        vm_offset_t         offset,
        pointer_t           data,
        mk_rpc_size_t       dataSize,
        boolean_t           dirty,
        boolean_t           kernel_copy);
```

DESCRIPTION

The kernel invokes **memory_object_data_return** to push data that has been evicted from the physical memory cache to that data's memory manager. The kernel writes back only data that has been modified or is precious.

PARAMETERS

abs_mem_port

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

mem_cache_control

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

offset

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

data

[in pointer to dynamic array of bytes] The data that has been evicted from the physical memory cache; must be on a page boundary.

dataSize

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

dirty

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

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

NOTES

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

RETURN VALUE

RELATED INFORMATION

Functions: **memory_object_data_supply**.

memory_object_data_unlock

Server Interface — Requests access to a memory object

SYNOPSIS

```
kern_return_t memory_object_data_unlock
        (mach_port_t        abs_mem_port,
        mach_port_t         mem_cache_control,
        vm_offset_t         offset,
        vm_size_t           length,
        vm_prot_t           desired_access);
```

DESCRIPTION

The kernel invokes **memory_object_data_unlock** to request permission to access cached data that is currently protected. The memory manager is expected to use the **memory_object_lock_request** call in response.

PARAMETERS

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

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

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

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

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

  VM_PROT_READ
  : Allows read access.

VM_PROT_WRITE
: Allows write access.

VM_PROT_EXECUTE
: Allows execute access.

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_lock_completed**, **memory_object_lock_request**.

memory_object_lock_completed

Server Interface — Indicates completion of a consistency control call

SYNOPSIS

```
kern_return_t memory_object_lock_completed
        (mach_port_t                    notify_port,
        mach_port_t                     mem_cache_control,
        vm_offset_t                     offset,
        vm_size_t                       length);
```

DESCRIPTION

The kernel invokes **memory_object_lock_completed** to confirm its 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:

- Modify 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.

The kernel invokes **memory_object_lock_completed** on the notification port explicitly provided in the corresponding **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** notification 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

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

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

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

length

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_lock_request**.

memory_object_notify

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

SYNOPSIS

```
kern_return_t memory_object_notify
        (mach_port_t        mem_obj_rep,
        mach_port_t         mem_cache_control,
        mach_port_t         host_name,
        vm_size_t           page_size);
```

DESCRIPTION

The kernel invokes **memory_object_notify** to inform a memory manager that the kernel has been asked to map the specified memory object representative into a task's virtual address space. The kernel invokes **memory_object_notify** the first time it is asked to map the corresponding object into an address space; the invocation of **memory_object_notify** initiates hand-shake dialog authenticating the memory manager (the receiver for the representative port) and the kernel to each other, thereby establishing the identity and properties of the underlying memory object.

The call supplies the memory manager with the kernel-generated memory cache control port, which the memory manager specifies in a subsequent **memory_object_establish** call to authenticate itself to the kernel; the memory manager continues to use this port to manipulate the memory cache object once initialization is complete. The supplied *host_name* port identifies the kernel mapping the object. The memory manager uses this port to verify the identity of the kernel issuing the **memory_object_notify** RPC. This port is the destination of the subsequent **memory_object_establish** RPC.

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

The memory cache control port is initially created with default attributes. It (as well as the associated memory cache object) will be assigned its proper attributes when the kernel handles the memory manager's invocation of **memory_object_establish**. The memory manager should refrain from invoking RPCs via the memory cache control port until memory object initialization is complete.

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

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

PARAMETERS

mem_obj_rep

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

mem_cache_control

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

host_name

[in host-name send right] The name port for the host requesting the memory object initialization. (The function **mach_host_self** returns send rights to the host's name port.)

page_size

[in scalar] The page size used by the kernel. All calls involving this kernel must use data sizes that are integral multiples of this page size and offsets that fall on page boundaries.

NOTES

When the memory manager is ready to manage this object, it verifies that the supplied host name port does name a host and then uses this port to invoke the **memory_object_establish** RPC.

When the kernel invokes **memory_object_notify**, it is informing the memory manager that it (the kernel) has no means of associating the memory object representative with the underlying abstract memory object. If the abstract object underlying the memory representative is currently initialized, the manager should respond with **memory_object_representative** instead of **memory_object_establish**, since the supplied memory cache control port does not name the memory cache object for the abstract memory object.

RETURN VALUE

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

RELATED INFORMATION

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

memory_object_rejected

Server Interface — Indicates that a **memory_object_establish** RPC was ignored

SYNOPSIS

```
kern_return_t memory_object_rejected
        (mach_port_t                    abs_mem_port,
        mach_port_t                     rejected_cache_control,
        kern_return_t                   reason);
```

DESCRIPTION

The kernel invokes **memory_object_rejected** to inform a memory manager that the kernel has rejected the manager's previous invocation of the **memory_object_establish** RPC. This occurs when the manager believes it has finished initializing a memory object that either was not being initialized (a client faked a **memory_object_notify** RPC) or the kernel no longer wishes to initialize the object.

The kernel invokes **memory_object_rejected** to allow the memory manager to clean up data structures associated with the invalid mappings. In such cases, the *memory_control* port sent isn't really a memory cache control port. The kernel returns send rights to this port as a means of identifying which **memory_object_establish** RPC was ignored.

PARAMETERS

abs_mem_port

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

rejected_cache_control

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

reason

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

KERN_INVALID_MEMORY_CONTROL

The memory cache control port supplied in the corresponding **memory_object_establish** RPC was not a memory cache control port.

KERN_INVALID_LEDGER
: The ledger port supplied in the corresponding **memory_object_establish** RPC was not a ledger port.

RETURN VALUE

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

RELATED INFORMATION

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

memory_object_supply_completed

Server Interface — Indicates completion of a data supply call

SYNOPSIS

```
kern_return_t memory_object_supply_completed
        (mach_port_t        abs_mem_port,
        mach_port_t         mem_cache_control,
        vm_offset_t         offset,
        vm_size_t           length,
        kern_return_t       result,
        vm_offset_t         error_offset);
```

DESCRIPTION

The kernel invokes **memory_object_supply_completed** to confirm the kernel's action in response to a **memory_object_data_supply** RPC from the memory manager.

The kernel invokes **memory_object_supply_completed** on the notification port explicitly provided by the memory manager via the **memory_object_data_supply** RPC. 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 RPC, it will indicate so in the completion response. If the pages not accepted are precious, they will be returned (via **memory_object_data_return** RPCs) before the kernel invokes **memory_object_supply_completed**. The completion RPC includes the offset and length values from the supply request to distinguish it from other supply requests.

PARAMETERS

abs_mem_port

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

mem_cache_control

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

offset

[in scalar] The offset within the memory object from the corresponding data supply call. The offset must be on a page boundary.

length

[in scalar] The number of bytes accepted. The number converts to an integral number of memory object pages.

result

[in scalar] A kernel return code indicating the result of the supply operation, possibly KERN_SUCCESS. KERN_MEMORY_PRESENT is currently the only recoverable error returned; other errors (invalid arguments, for example) abort the data supply operation.

error_offset

[in scalar] The offset within the memory object where the first error occurred.

RETURN VALUE

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

RELATED INFORMATION

Functions: **memory_object_data_supply**.

memory_object_terminate

Server Interface — Relinquishes access to a memory object

SYNOPSIS

```
kern_return_t memory_object_terminate
        (mach_port_t                    abs_mem_port,
        mach_port_t                     mem_cache_control);
```

DESCRIPTION

The kernel invokes **memory_object_terminate** to inform a memory manager that no mappings of the specified memory object remain. The kernel invokes this RPC to allow the memory manager to clean up data structures associated with the de-allocated mappings. The kernel relinquishes its rights for all memory object ports.

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

PARAMETERS

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

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

RETURN VALUE

RELATED INFORMATION

Functions: **memory_object_destroy**, **memory_object_notify**, **mach_port_deallocate**.

CHAPTER 13 Privileged Interfaces

This chapter discusses “privileged” kernel calls — those that can affect the protection of the system.

add_partition

Function — Adds a partition to the default memory manager's backing store.

SYNOPSIS

```
kern_return_t add_partition
        (mach_port_t    service,
        device_t        device,
        recnum_t        recnum,
        vm_size_t       recsize,
        vm_size_t       length);
```

DESCRIPTION

The **add_partition** function provides the identity of backing space to the default memory manager.

PARAMETERS

service
: [in default-pager send right] The service port to the default memory manager, as returned by **bootstrap_ports**.

device
: [in device send right] The port naming the backing store device.

recnum
: [in scalar] The record number of the first record to use.

recsize
: [in scalar] The size, in bytes, of each record on the device.

length
: [in scalar] The total size of the partition, in bytes.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**, **bootstrap_ports**.

bootstrap_arguments

Function — Obtain arguments used by the bootstrap task to invoke servers

SYNOPSIS

```
kern_return_t bootstrap_arguments
        (mach_port_t                    bootstrap,
        boot_args_t                     arguments,
        mk_rpc_size_t*                  argumentsCnt);
```

DESCRIPTION

The **bootstrap_arguments** function obtains the arguments that were used by the bootstrap task to invoke servers.

PARAMETERS

bootstrap

[in bootstrap send right] A bootstrap port, the host's or that obtained by **bootstrap_ports**.

arguments

[out string] The argument string provided in the "call" of the user task at its invocation by the bootstrap task.

argumentsCnt

[out scalar] The number of space-separated items in *arguments*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **bootstrap_ports**

bootstrap_ports

Function — Obtains selected system privilege ports from the kernel or a bootstrap task

SYNOPSIS

```
kern_return_t bootstrap_ports
        (mach_port_t            bootstrap,
        boot_port_mask_t*       port_mask,
        mach_port_array_t       boot_ports);
```

DESCRIPTION

The **bootstrap_ports** function returns send rights to the requested special ports, as indicated by *port_mask*.

PARAMETERS

bootstrap

[in bootstrap send right] The bootstrap port obtained by **task_get_bootstrap_port**.

port_mask

[pointer to in/out scalar] A mask obtained by a bitwise OR of the following values:

GET_DEFAULT_PAGER_PORT

Return the default pager's service port (not applicable to the kernel; applicable to the system bootstrap task)

GET_DEVICE_MASTER_PORT

Return the device master port; used with **device_open**.

GET_HOST_CONTROL_PORT

Return the host control port; used to obtain other privileged ports and information.

GET_HOST_PAGING_PORT

Return the host paging port; used with **vm_set_default_memory_manager**.

GET_HOST_SECURITY_PORT

Return the host security port; used for **task_create_security_id** and **host_security_id_list**.

GET_LEDGER_CONTROL_PORT
: Return the control port for the ledger containing the requesting task's resources (not applicable to the kernel).

boot_ports

[out array of send rights] The returned port right, indexed as follows (all unreturned rights are NULL):

DEFAULT_PAGER_PORT_INDEX

DEVICE_MASTER_PORT_INDEX

HOST_CONTROL_PORT_INDEX

HOST_PAGING_PORT_INDEX

HOST_SECURITY_PORT_INDEX

LEDGER_CONTROL_PORT_INDEX

RETURN VALUE

Only generic errors apply. The port mask is adjusted at output to indicate the ports actually delivered.

RELATED INFORMATION

Functions: **device_open**, **host_get_clock_control**, **host_processor_set_priv, task_create_security_id**, **host_security_id_list**, **task_get_special_port, vm_set_default_memory_manager**.

device_open

Function — Establish a connection to a device.

SYNOPSIS

```
kern_return_t device_open
        (mach_port_t        master_port,
        mach_port_t         ledger,
        dev_mode_t          mode,
        security_id_t       security_id,
        dev_name_t          name,
        mach_port_t*        device_port);
```

DESCRIPTION

The **device_open** function opens a device object. The open operation of the device driver is invoked. The open count for the device is incremented. Each open for a device returns a distinct port, the allowed operations upon which being governed by *mode*.

PARAMETERS

master_port
: [in device-master send right] The device master port. This port is provided to/by the bootstrap task.

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

mode
: [in scalar] Opening mode. This is the bit-wise OR of the following values:

  D_EXCLUSIVE
  : Exclusive use of the device

  D_NODELAY
  : Do not delay on open

  D_READ
  : Read access

  D_VOLUME
  : Allow access to volume label and beyond disk partition limits

D_WRITE
Write access

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

name
[pointer to in array of *char*] Name of the device to open.

device_code
[in scalar] Status of the open.

device_port
[out device send right, in for asynchronous form] The returned device port.

RETURN VALUE

The error return from **device_open** is one of the following:

D_WOULD_BLOCK
The device is busy, but D_NODELAY was specified in *mode*.

D_NO_SUCH_DEVICE
No device with that name, or the device is not operational.

D_DEVICE_DOWN
The device has been shut down.

D_READ_ONLY
Data cannot be written to this device.

KERN_INVALID_LEDGER
ledger was not a kernel ledger.

RELATED INFORMATION

Functions: **device_close**.

host_get_boot_info

Function — Return operator boot information

SYNOPSIS

kern_return_t **host_get_boot_info**
(mach_port_t *host_priv,*
kernel_boot_info_t *boot_info*);

DESCRIPTION

The **host_get_boot_info** function returns the boot-time information string supplied by the operator when *host_priv* was initialized. The constant KERNEL_BOOT_INFO_MAX (in **mach/host_info.h**) should be used to dimension storage for the returned string.

PARAMETERS

host_priv

[in host-control send right] The control port for the host for which information is to be obtained.

boot_info

[out array of *char*] Character string providing the operator boot information.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**.

host_get_clock_control

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

SYNOPSIS

```
kern_return_t host_get_clock_control
        (mach_port_t        host_priv,
        clock_id_t          clock_id,
        mach_port_t*        clock_control);
```

DESCRIPTION

The **host_get_clock_control** function returns a send right to the control port for a kernel clock object. This right is used to set the clock's resolution and time.

PARAMETERS

host_priv

[in host-control send right] The control port for the host owning the clock.

clock_id

[in scalar] The identification of the desired kernel clock. These values are defined in **<mach/clock_types.h>**. Although an implementation may define additional values, the following values are always defined (although only the REALTIME clock is required to be implemented):

REALTIME_CLOCK

A moderate resolution clock service that (typically) tracks time since the system last boot.

BATTERY_CLOCK

A (typically) low resolution clock (to the second) that survives power failures or service outages.

HIGHRES_CLOCK

A high resolution clock.

clock_control

[out clock-control send right] Control port for the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **clock_set_time**, **clock_set_attributes**, **host_get_clock_service**.

host_processor_set_priv

Function — Translates a processor set name port into a processor set control port

SYNOPSIS

kern_return_t **host_processor_set_priv**
(mach_port_t *host_priv,*
mach_port_t *set_name,*
mach_port_t* *processor_set*);

DESCRIPTION

The **host_processor_set_priv** function returns send rights for the control port for a specified processor set currently existing on *host_priv.*

PARAMETERS

host_priv
[in host-control send right] The control port for the host for which the processor set is desired.

set_name
[in processor-set-name send right] The name port for the processor set desired.

processor_set
[out processor-set-control send right] The returned processor set control port.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **processor_set_default, processor_set_policy_control**.

host_reboot

Function — Reboot this host

SYNOPSIS

kern_return_t **host_reboot**
(mach_port_t *host_priv,*
host_reboot_t *options*);

DESCRIPTION

The **host_reboot** function reboots the specified host.

PARAMETERS

host_priv
[in host-control send right] The control port the host to be re-booted.

options
[in scalar] Reboot options. See **<sys/reboot.h>** for details.

NOTES

If successful, this call will not return.

RETURN VALUE

Only generic errors apply.

host_sample

Function — Perform periodic PC sampling for a host

SYNOPSIS

```
kern_return_t host_sample
        (mach_port_t        host_priv,
        mach_port_t         reply_port);
```

DESCRIPTION

The **host_sample** function causes the program counter (PC) of the specified *host* (actually, all of the processors within the host) to be sampled periodically. The set of PC sample values obtained are saved in buffers which are sent to the specified *reply_port* in **receive_samples** messages. Only PC values within the kernel are reported; to profile user tasks, **task_sample** and **thread_sample** are used instead.

PARAMETERS

host_priv

[in host-control send right] The control port for the host for which information is to be obtained.

reply_port

[in sample receive (to be converted to send) right] Port to which PC sample buffers are sent. A value of MACH_PORT_NULL stops PC sampling for the host.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **end_samples**, **receive_samples**.

host_security_id_list

Function — Returns a list of the active security IDs

SYNOPSIS

```
kern_return_t host_security_id_list
        (mach_port_t                host_security,
        security_id_array_t         security_id_list,
        mach_msg_type_number_t*     security_id_listCnt);
```

DESCRIPTION

The **host_security_id_list** function returns a list of the active security IDs within *host_security.* A security ID is active if at least one existing task has that security ID.

PARAMETERS

host_security
: [in security send right] The security port for the host for which the security ID list is to be returned, obtained via **bootstrap_ports**.

security_id_list
: [out array of security IDs] The returned list of security IDs, in no particular order.

security_id_listCnt
: [in/out scalar] On input, the size of *security_id_list*; on output, the returned count of security IDs in *security_id_list*.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create_security_id**, **mk_rpc**, **bootstrap_ports**.

host_statistics

Function — Returns statistics for a host

SYNOPSIS

```
kern_return_t host_statistics
        (mach_port_t        host_priv,
        host_flavor_t       flavor,
        host_info_t         host_info_out,
        mk_rpc_size_t*      host_info_outCnt);
```

DESCRIPTION

The **host_statistics** function returns scheduling and virtual memory statistics concerning the host as specified by *flavor*.

PARAMETERS

host_priv
: [in host-control send right] The control port for the host for which information is to be obtained.

flavor
: [in scalar] The type of statistics desired.

HOST_LOAD_INFO
: System loading statistics. The returned structure is **host_load_info**.

HOST_VM_INFO
: Virtual memory statistics. The returned structure is **vm_statistics**.

host_info_out
: [out structure] Statistics about the specified host.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_info**, **processor_set_statistics**.

ledger_create

Function — Create a new ledger

SYNOPSIS

```
kern_return_t ledger_create
        (mach_port_t                host_priv,
        resource_usage_t            memory_entitlement,
        resource_ratio_t            memory_ratio,
        resource_usage_t            virtual_entitlement,
        resource_ratio_t            virtual_ratio,
        mach_port_t*                ledger_control,
        mach_port_t*                ledger_name);
```

DESCRIPTION

The **ledger_create** function creates a new ledger with specified resource limits.

PARAMETERS

host_priv
: [in host-control send right] The control port for the host.

memory_entitlement
: [in scalar] The amount of memory (in bytes) that can be used regardless of system load (applies to wired memory).

memory_ratio
: [in scalar] The ratio of system resources granted (applies to wired memory).

virtual_entitlement
: [in scalar] The amount of memory (in bytes) that can be used regardless of system load (applies to address space size).

virtual_ratio
: [in scalar] The ratio of system resources granted (applies to address space size).

ledger_control
: [out ledger-control send right] The control port for the new ledger.

ledger_name
: [out ledger-name send right] The name port for the new ledger.

NOTES

The kernel sporadically checks all of the system ledgers. If the resource consumption associated with a ledger exceeds the specified entitlement and the ratio of system resources consumed exceeds the specified ratio, then the ledger is considered for destruction, destroying all objects bound to it.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **ledger_terminate**, **task_create**, **memory_object_establish**, **device_open**, **clock_alarm**.

task_create_security_id

Function — Creates a new task with an explicit security ID

SYNOPSIS

```
kern_return_t task_create_security_id
        (mach_port_t        host_security,
        mach_port_t         ledger_name,
        mach_port_t         parent_task,
        security_id_t       security_id,
        boolean_t           inherit_memory,
        mach_port_t*        child_task);
```

DESCRIPTION

The **task_create_security_id** function creates a new task from *parent_task* with an explicit security ID, returning the name of the new task in *child_task*. Other than the security ID, the child task is as if created by **task_create**.

PARAMETERS

host_security
: [in security send right] The security port for the host, obtained via **bootstrap_ports**.

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

parent_task
: [in task send right] The port for the task from which to draw the child task's port rights and address space.

security_id
: [in scalar] The task's security ID.

inherit_memory
: [in boolean] Address space inheritance indicator. If true, the child task inherits the address space of the parent task. If false, the kernel assigns the child task an empty address space.

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

NOTES

The host security port is a privileged port given to the system bootstrap task for the use of this call.

RETURN VALUE

KERN_INVALID_LEDGER
: *ledger* does not name a ledger on the host on which *task* lies.

KERN_INVALID_TASK
: *parent_task* does not name a task.

RELATED INFORMATION

Functions: **task_create**, **mk_rpc,** **server_dispatch**, **host_security_id_list**, **bootstrap_ports**.

thread_wire

Function — Marks the thread as privileged with respect to kernel resources

SYNOPSIS

```
kern_return_t thread_wire
        (mach_port_t        host_priv,
        mach_port_t         target_thread,
        boolean_t           wired);
```

DESCRIPTION

The **thread_wire** function marks the thread as "wired". A "wired" thread is always eligible to be scheduled and can consume physical memory even when free memory is scarce. This property should be assigned to threads in the default page-out path. Threads not in the default page-out path should not have this property to prevent the kernel's free list of pages from being exhausted.

PARAMETERS

host_priv
: [in host-control send right] The privileged control port for the host on which the thread executes.

target_thread
: [in thread send right] The thread to be wired.

wired
: [in scalar] TRUE if the thread is to be wired.

RETURN VALUE

KERN_INVALID_THREAD
: *target_thread* does not name a thread.

RELATED INFORMATION

Functions: **vm_wire**.

vm_set_default_memory_manager

Function — Sets the default memory manager.

SYNOPSIS

```
kern_return_t vm_set_default_memory_manager
        (mach_port_t        virtual_paging_object,
        mach_port_t         backing_space);
```

DESCRIPTION

The **vm_set_default_memory_manager** function establishes the abstract backing space object (the default memory manager) for a host.

PARAMETERS

virtual_paging_object
: [in virtual-paging-object send right] The port naming the kernel's virtual paging object, obtained via **bootstrap_ports**.

backing_space
: [in abstract-backing-space send right] A memory manager port naming the abstract backing space object to which anonymous pages will be sent (actually, the backing space object to which the kernel will send page-in and page-out requests).

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **vm_allocate**, **vm_set_recycle_limit**, **add_partition**, **bootstrap_ports**.

vm_set_recycle_limit

Function — Adjust kernel's paging space recycle threshold

SYNOPSIS

```
kern_return_t vm_set_recycle_limit
        (mach_port_t        virtual_paging_object,
        vm_size_t           recycle_limit);
```

DESCRIPTION

The **vm_set_recycle_limit** function establishes the kernel's backing space recycle threshold.

The allocation of blocks in the virtual backing space is performed by the kernel. The kernel has a tendency to stream data out, allocating further and further into the backing space. Setting the recycle limit affects the threshold before the kernel starts reallocating space.

PARAMETERS

virtual_paging_object
: [in virtual-paging-object send right] The port naming the kernel's virtual paging object, obtained via **bootstrap_ports**.

recycle_limit
: [in scalar] The new recycle limit, in bytes.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **vm_set_default_memory_manager**, **bootstrap_ports**.

CHAPTER 14

Protected Interfaces

This chapter discusses protected, or restricted, interfaces. Use of these interfaces does not affect the protection afforded by the system but do affect the operation of the system.

clock_set_attributes

Function — Sets attributes of a clock

SYNOPSIS

```
kern_return_t clock_set_attributes
        (mach_port_t                clock_ctrl,
        clock_flavor_t              flavor,
        clock_attr_t                clock_attr,
        mach_msg_type_number_t      clock_attrCnt);
```

DESCRIPTION

The **clock_set_attributes** function sets attributes of a clock's operation.

PARAMETERS

clock_ctrl

[in clock-control send right] The control port for the clock.

flavor

[in scalar] Type of information to be set. Defined values are:

CLOCK_ALARM_CURRES

The resolution, in nanoseconds, at which clock alarm and sleep timers are currently serviced. Increasing the current resolution will have no impact on any pending clock alarms (i.e. they will go off as originally scheduled). Decreasing the current resolution will truncate any pending alarms to the granularity of the new current resolution. This value must be a multiple of the minimum resolution (the smallest interval between clock ticks) and not greater than the maximum resolution of the clock.

clock_attr

[pointer to in scalar] New attribute.

clock_attrCnt

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

NOTES

The main reason a clock's current resolution would not always equal its minimum resolution is because the overhead of sustaining the minimum resolution, when it is not needed by any existing alarm service client, may be prohibitive for a given hardware platform and underlying clock device.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_control**, **clock_set_time**, **clock_get_attributes**.

clock_set_time

Function — Sets the current time

SYNOPSIS

kern_return_t **clock_set_time**
(mach_port_t *clock_ctrl,*
tvalspec_t *new_time*);

DESCRIPTION

The **clock_set_time** function sets the time kept by a clock. Setting the clock time will cause all pending clock alarms and sleeps to be terminated with time-stamps set to the current clock time just prior to the new time being set with a return code of KERN_ABORTED.

PARAMETERS

clock_ctrl
[in clock-control send right] The control port for the clock.

new_time
[in structure] The new time.

CAUTIONS

The use of this function is highly discouraged since it could affect the monotonically increasing nature of the clock.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **host_get_clock_control**, **clock_set_attributes**, **clock_get_time**, **alarm_set**, **clock_sleep**.

Data Structures: **tvalspec**.

ledger_terminate

Function — Destroy a ledger

SYNOPSIS

kern_return_t **ledger_terminate**
(mach_port_t *target_ledger*);

DESCRIPTION

The **ledger_terminate** function destroys a ledger. All resources drawn from this ledger will also be destroyed.

PARAMETERS

target_ledger
[in ledger-control send right] The ledger to destroy.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **ledger_create**.

processor_set_policy_control

Function — Controls the scheduling attributes governing the processor set

SYNOPSIS

kern_return_t **processor_set_policy_control**
(mach_port_t *processor_set_control,*
processor_set_flavor_t *flavor,*
processor_set_info_t *policy_info*,
mk_rpc_size_t *policy_infoCnt*,
boolean_t *change_task_threads*);

DESCRIPTION

The **processor_set_policy_control** function controls scheduling attributes governing the processor set.

PARAMETERS

processor_set_control
[in processor-set-control send right] A processor set control port.

flavor
[in scalar] The type of policy change to make.

PROCESSOR_SET_TIMESHARE_DEFAULT
Change the base attributes for the timeshare scheduling policy, making timeshare the default policy. The structure is **policy_timeshare_base**.

PROCESSOR_SET_FIFO_DEFAULT
Change the base attributes for the FIFO (first-in, first-out) scheduling policy, making FIFO the default policy. The structure is **policy_fifo_base**.

PROCESSOR_SET_RR_DEFAULT
Changed the base attributes for the round-robin scheduling policy, making round robin the default policy. The structure is **policy_rr_base**.

PROCESSOR_SET_TIMESHARE_LIMITS
Change the limits on the allowed timeshare policy attributes. The structure is defined by **policy_timeshare_limit**.

PROCESSOR_SET_RR_LIMITS
Change the limits on the allowed round robin policy attributes. The structure is defined by **policy_rr_limit**.

PROCESSOR_SET_FIFO_LIMITS
: Change the limits on the allowed first-in, first-out policy attributes. The structure is defined by **policy_fifo_limit**.

PROCESSOR_SET_ENABLED_POLICIES
: Change the set of enabled policies. The data is a bit-vector.

policy_info
: [in structure] The relevant policy information.

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

change_task_threads
: [in boolean] If true, any assigned task or thread whose policy is no longer enabled or whose scheduling attributes exceed the current limits will have their limits adjusted or their policy set to the default as appropriate.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**:** **processor_set_statistics**, **processor_set_default**, **processor_set_info**.

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

processor_set_statistics

Function — Returns scheduling statistics for a processor set.

SYNOPSIS

```
kern_return_t processor_set_statistics
        (mach_port_t                processor_set_control,
        processor_set_flavor_t      flavor,
        processor_set_info_t        pset_info,
        mk_rpc_size_t*              pset_infoCnt);
```

DESCRIPTION

The **processor_set_statistics** function returns statistics for a processor set as specified by *flavor*.

PARAMETERS

processor_set_control
: [in processor-set-control send right] A processor set control port for which information is desired.

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

  PROCESSOR_SET_LOAD_INFO
  : Load statistics for the processor set. The returned structure is **processor_set_load_info**.

pset_info
: [out structure] Information about the processor set.

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions**: processor_set_info**.

processor_set_tasks

Function — Returns a list of the tasks bound to a processor set

SYNOPSIS

```
kern_return_t processor_set_tasks
        (mach_port_t                processor_set_control,
        mach_port_array_t           tasks,
        mk_rpc_size_t *             tasksCnt);
```

DESCRIPTION

The **processor_set_tasks** function returns a list of all tasks currently bound to the processor set.

PARAMETERS

processor_set_control

[in processor-set-control send right] The control port for the processor set.

tasks

[out array of task send rights] The returned list of tasks within *processor_set_control*, in no particular order.

tasksCnt

[in/out scalar] On input, the size of *tasks*; on output, the returned count of tasks in *tasks*. The maximum size of *tasks* is PROCESSOR_SET_TASKS_MAX.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **task_create**, **task_set_policy**.

task_set_policy

Function — Sets the scheduling policy to apply to a task

SYNOPSIS

```
kern_return_t task_set_policy
        (mach_port_t        processor_set_control,
        mach_port_t         task,
        policy_t            policy,
        policy_base_t       base,
        mk_rpc_size_t       baseCnt,
        policy_limit_t      limit,
        mk_rpc_size_t       limitCnt,
        boolean_t           change);
```

DESCRIPTION

The **task_set_policy** function sets the scheduling attributes, both base and limit, for *task*. *policy* may be any policy implemented by the processor set whether or not it is enabled.

PARAMETERS

processor_set_control

[in processor-set-control send right] The control port for the processor set to which the task is currently assigned.

task

[in task send right] The task whose scheduling policy is to be set.

policy

[in scalar] Policy to be set. The values currently defined are POLICY_TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base

[pointer to in structure] Base policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

baseCnt

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

limit

[pointer to in structure] Policy specific limits, **policy_fifo_limit**, **policy_rr_limit** or **policy_timeshare_limit**.

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

change
[in boolean] True if the scheduling attributes for all contained threads should be changed as well.

RETURN VALUE

KERN_INVALID_TASK
task is not a task in processor set *processor_set_control*.

RELATED INFORMATION

Functions: **processor_set_policy_control**, **thread_policy**, **thread_set_policy**, **task_policy**.

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

thread_set_policy

Function — Sets the scheduling policy to apply to a thread

SYNOPSIS

```
kern_return_t thread_set_policy
        (mach_port_t        processor_set_control,
        mach_port_t         thread,
        policy_t            policy,
        policy_base_t       base,
        mk_rpc_size_t       baseCnt,
        policy_limit_t      limit,
        mk_rpc_size_t       limitCnt);
```

DESCRIPTION

The **thread_set_policy** function sets the scheduling attributes, both base and limit, for *thread*. *policy* may be any policy implemented by the processor set whether or not it is enabled.

PARAMETERS

processor_set_control

[in processor-set-control send right] The control port for the processor set to which the thread is currently assigned.

thread

[in thread send right] The thread scheduling policy is to be set.

policy

[in scalar] Policy to be set. The values currently defined are POLICY_TIMESHARE, POLICY_RR (round robin) and POLICY_FIFO (first-in, first-out).

base

[pointer to in structure] Base policy specific data, **policy_fifo_base**, **policy_rr_base** or **policy_timeshare_base**.

baseCnt

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

limit

[pointer to in structure] Policy specific limits, **policy_fifo_limit**, **policy_rr_limit** or **policy_timeshare_limit**.

limitCnt

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

RETURN VALUE

KERN_INVALID_THREAD
: *thread* is not a thread in processor set *processor_set_control*.

RELATED INFORMATION

Functions: **processor_set_policy_control**, **thread_policy**, **task_policy**, **task_set_policy**.

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

APPENDIX A

Intel Pentium Platform Device Interface Specifics

This appendix supplements the Kernel Device Interface chapter by providing specific argument values and constraints associated with the device drivers available for the Intel Pentium (and Pentium Pro) reference platform. Only PCI-based systems are supported in the reference implementation.

Device Names

The general form for MK++ device names is **aabc**, where **aa** represents the device class, **b** is the unit number within that class, and **c** is an optional subunit or partition identifier. The unit number is numeric, starting at zero; the subunit/partition is alphabetic, starting with "`a`" and limited by the respective device driver. Most physical devices carry at least two such names, the first of which is computed from the device driver (e.g. "`sd`" for SCSI Disk) and unit number. An alias is also assigned by generic class (e.g. "`disk`") with a system-wide sequential unit number.

For the purposes of the **device_open** call, **b** defaults to zero and **c** defaults to null (implying the first subunit). As a result, the first partition of the first configured disk drive may be referred to as "`disk`", "`disk0`", or "`disk0a`" interchangeably.

Device Concurrency and Covert Storage Channels

While the MK++ kernel is explicitly *not* the reference monitor for device access, it is called upon to enforce some per-request constraints upon shared devices. The general requirement is that an untrusted reader of a shared device shall not be capable of modifying device state in any way that is detectable by another reader of the same device. For this reason, certain operations that might otherwise be permissible to a reader now require one or both of the new `D_EXCLUSIVE` or `D_VOLUME` flags to have been specified to **device_open**. Some devices are necessarily unshareable among concurrent users, which simplifies such checks in those cases.

Console, Keyboard, and Mouse Drivers

The physical console is named "kd0" and is configured as the first device on the system. It acquires "tty0" as its alias by virtue of its character-mode operation. When the cons_is_com1 boot flag is *not* set, the special "console" alias is also applied. The (raw) keyboard, PS2 mouse, and PORT mouse devices are assigned "kd0b", "kd0c", and "kd0d", respectively.

In addition to being identified as console subaddresses, the keyboard and various mouse devices have been given the following aliases for use by the X server:

- "kbd0" is the console keyboard in "raw" mode.
- "kbd2" is a PS2 mouse, if configured.
- "kbd3" is a port mouse, if configured.
- "kbd4" is a (two-button) Microsoft® mouse on COM1.
- "kbd5" is a (three-button) Mouse Systems® mouse on COM1.
- "kbd6" is a (two-button) Microsoft® mouse on COM2.
- "kbd7" is a (three-button) Mouse Systems® mouse on COM2.

The console driver is configured at physical and port address 0x60 and is assigned Processor Interrupt Channel 1. Console interrupts are handled at the SPL_TTY level.

Open Modes and Concurrency

The "raw" keyboard (kd0b or kb0d) is opened exclusively, as is whichever mouse is configured. The console display is sharable among multiple writers.

Opening modes D_WRITE and D_READ may be specified with all subdevices; no checks are made during the processing of **device_open**. Opening mode D_EXCLUSIVE should *not* be used with the physical console or its subdevices. Mode D_VOLUME is not meaningful, but will not be rejected.

Get Status Request Flavors

The following device_flavor_t values apply to the physical console/keyboard:

- KDGKBDTYPE returns the constant KB_VANILLAKB. It may only be specified on calls to the "raw" console keyboard device.
- KDGKBENT gets an entry from the current keyboard mapping table.
- KDGSTATE returns the current state of the keyboard as an integer.

See the section following for additional flavors that may be passed through to mouse devices connected to serial ports.

Set Status Request Flavors

The following device_flavor_t values apply to the physical console/keyboard:

- KDSETBELL enables/disables the speaker port, which produces the audible "beep" in response to BEL characters sent to the display.

- KDSKBDMODE sets the mode of the console keyboard to either KB_EVENT or KB_ASCII. It may only be specified on calls to the "raw" console keyboard device.
- KDSKBENT sets an entry in the current keyboard mapping table. D_WRITE mode is required.

See the section following for additional flavors that may be passed through to mouse devices connected to serial ports.

Unsupported Calls

Neither the **device_map** nor the **device_set_filter** calls are supported by the console driver.

Serial Communications (TTY) Drivers

Each serial communications port is named sequentially beginning with "com0" and has an alias of "tty#", where # is assigned sequentially to each configured tty device.

The serial communications ports are configured at port addresses 0x3f8, 0x2f8 0x3e8, and 0x2e8. They are assigned Processor Interrupt Channels 4, 3, 5, and 12, respectively. Note that the PICs for com2 and com3 can conflict with those of the configured Ethernet controller, below. Interrupts are handled at the SPL_TTY level.

Get Status Request Flavors

The following device_flavor_t values apply to character devices, including serial mouse devices associated with the system console.

- TTY_MODEM obtains modem control information for the serial port.
- TTY_OUT_OF_BAND reads out-of-band events from the serial device.
- TTY_STATUS_COMPAT obtains information about the serial device.
- TTY_STATUS_NEW obtains information about the serial device.

Set Status Request Flavors

The following device_flavor_t values apply to the character devices, including serial mouse devices associated with the system console.

- TTY_DRAIN discards queued output.
- TTY_FLUSH clears both input and output queues.
- TTY_START starts queued output.
- TTY_STOP stops output.
- TTY_SET_BREAK sets the break condition. D_WRITE mode is required.
- TTY_CLEAR_BREAK clears the break condition. D_WRITE mode is required.
- TTY_MODEM sets modem control information for the serial port. D_WRITE mode is required.
- TTY_NMODEM sets the soft carrier for the serial port. D_WRITE mode is required.
- KDSKBENT sets an entry in the current keyboard mapping table. D_WRITE mode is required. Applies only to the the console device when remapped onto COM1.
- TTY_STATUS_COMPAT sets special characters and speed for the serial device. D_WRITE mode is required.
- TTY_STATUS_NEW sets special characters and speed for the serial device. D_WRITE mode is required.

Unsupported Calls

Neither the **device_map** nor the **device_set_filter** calls are supported by the serial communications driver.

VGA Drivers

When configured, the single VGA monitor is known as "`vga0`". For compatibility with the X server, the "`iopl`" alias is assigned to the graphics card to enable device mapping.

Four VGA drivers are provided, which are probed at kernel initialization in the order listed below. If any of the first three drivers recognizes its supported graphics card, that driver is configured. The default is the generic driver for any other VGA card.

- The PVGA driver supports various graphics cards based upon the Western Digital 90C00 and 90C10 chips.
- The ET4000 driver supports the Sigma Legend and similar graphics cards.
- The S3 driver supports graphics cards from S3, Number 9 and Actix that use the S3 chip set.
- The generic VGA driver is used with all other graphics cards meeting basic VGA specifications.

The VGA controller is configured at port address 0x3b0 and physical address 0xa0000. It does not accept interrupts, so no Processor Interrupt Channel applies.

Device Map Specifics

Each VGA driver maps a 64KB window into the graphic card's video RAM under the control of the VGA I/O registers. Access to the device memory is constrained to not exceed the mode specified to **device_open** — write access to the mapped memory requires write access to the device.

For the Intel platform, **device_map** of a device open in `D_EXCLUSIVE` or `D_WRITE` mode also sets up access to the corresponding I/O registers. These registers are only usable while the memory object representative is mapped into the task's address space; they are activated by **vm_map** and deactivated by **vm_deallocate** of the region or **device_close** of the mapped device.

The following I/O registers apply to all VGA devices:

0x200	0x3b0 - 0x3df

The S3 driver defines the following additional I/O registers:

0x42e8 - 9	0x4ae8 - 9	0x82e8 - 9	0x86e8 - 9
0x8ae8 - 9	0x8ee8 - 9	0x92e8 - 9	0x96e8 - 9
0x9ae8 - 9	0x9ee8 - 9	0xa2e8 - b	0xa6e8 - b
0xaae8 - b	0xaee8 - b	0xb2e8 - b	0xb6e8 - 9
0xbae8 - 9	0xbee8 - 9	0xe2e8 - b	

Unsupported Calls

The VGA drivers do not support either the synchronous or asynchronous forms of **device_read** or **device_write**, nor is **device_set_filter** supported. Calls to **device_get_status** and **device_set_status** are ignored and return success status without performing any action.

Network Drivers

One network driver is provided, which supports multiple controllers. All interrupts are processed at the SPL_IMP interrupt level.

Multiple network interfaces based upon the Digital Tulip (DECchip 21040/21041) PCI Ethernet controller chipsets can be configured. The bus addresses and interrupt channels are assignd by the system PCI setup code. The assigned device names for Tulip controllers are "tu0" through "tu*N*".

Get Status Flavors

The following device_flavor_t values apply to the network devices.

- NET_ADDRESS returns an array of long integers that represent the network address of the device.
- NET_STATUS returns information on the network packet format used by this device.

Set Status Flavors

The following device_flavor_t value applies to the network devices.

- NET_STATUS sets selected flags to control the network device. The caller must have opened the device with D_WRITE permission.

Unsupported Calls

The network drivers do not support either the synchronous or asynchronous forms of **device_read**, since that functionality is performed through the use of device filters. The **device_map** service is also not supported.

Floppy Disk Driver

Floppy disk devices are named "fd0" and "fd1" when configured. Each drive is also assigned a "disk#" alias in the order of its recognition at system start-up. A subdevice suffix letter is used to indicate the media type, which defaults to "a".

- "fd0a" specifies a 3.5" DSDD (720KB) diskette.
- "fd0b" specifies a 3.5" DSHD (1.4MB) diskette.
- "fd0c" specifies a 5.25" DSSD (360KB) diskette.
- "fd0d" specifies a 5.25" DSHD (1.2MB) diskette.

The floppy controller is configured at port address 0x3f2 and assigned Processor Interrupt Channel 6. Floppy disk interrupts are handled at the SPL_BIO level.

Get Status Request Flavors

The following device_flavor_t values apply to floppy disk devices.

- DEV_GET_SIZE obtains the size of device blocks and the total size of the device.
- V_GETPARMS returns the media geometry information for the device.

Set Status Request Flavors

The following device_flavor_t values apply to floppy disk devices.

- V_FORMAT formats the volume. Caller must have D_WRITE, D_EXCLUSIVE, and D_VOLUME access to the device.
- V_SETPARMS causes the drive parameters to be rechecked. Caller must have D_WRITE access to the volume.
- V_VERIFY is provided for compatibility with other disk drivers but has no effect. Caller must have D_WRITE access to the volume.

Unsupported Calls

Neither the **device_map** nor the **device_set_filter** calls are supported by the floppy disk driver.

SCSI Subsystem

Both disk and tape device types are supported via Small Computer Systems Interface (SCSI) controllers. A driver has been provided for the Symbios Logic (NCR) 8xx PCI controller.

NCR 8xx PCI SCSI Controllers

Any number of NCR 810 or 825 controllers are supported. Interrupts are processed at the `SPL_BIO` interrupt level.

SCSI Disk Devices

For each configured controller, eight disk devices are configured at SCSI addresses 0 through 7. Disk units are named "`sd#`", where `#` is computed as 16 * the controller number (starting at zero) + the SCSI address of the unit. Disk units may have up to 16 partitions, identified sequentially from "`a`" to "`p`". For compatibility with non-SCSI systems, each unit is given an alias of "`hd#`" to match "`sd#`". Each disk device (of any type) also receives a "`disk#`" alias, where `#` is assigned sequentially across the entire system. The hard disk boot partition also receives the "`boot_device`" alias.

Get Status Request Flavors

The following `device_flavor_t` values apply to SCSI disk devices.

- `DEV_GET_SIZE` obtains the size of device blocks and the total size of the device.
- `DIOCGDINFO` reads the disk label.
- `V_GETPARMS` returns the media geometry information for the device.
- `V_RDABS` reads data using the absolute block number set with `V_ABS`. Caller must have opened the device with `D_VOLUME` access.
- `V_VERIFY` read/verifies data using the absolute block number set with `V_ABS`. Caller must have opened the device with `D_VOLUME` access.

Set Status Request Flavors

The following `device_flavor_t` values apply to SCSI disk devices.

- `DIOCSDINFO` sets the in-memory image of the disk label. Caller must have opened the device with `D_VOLUME` and `D_WRITE` access.
- `DIOCSRETRIES` updates the retry threshold for bad block replacement.
- `DIOCWDINFO` writes the disk label. Caller must have opened the device with `D_VOLUME` and `D_WRITE` access.
- `DIOCWLABEL` write-enables/disables the label. Caller must have opened the device with `D_VOLUME` and `D_WRITE` access.
- `V_ABS` sets the absolute block number for the next `V_RDABS`, `V_VERIFY`, or `V_WRABS` request. Caller must have opened the device with `D_VOLUME` access.
- `V_REMOUNT` remounts the drive. Caller must have opened the device with `D_VOLUME` access.
- `V_WRABS` writes data using the absolute block number set with `V_ABS`. Caller must have opened the device with `D_VOLUME` and `D_WRITE` access.

SCSI Tape Devices

For each configured controller, eight tape devices are configured at SCSI addresses 0 through 7. Tape units are named "`st#`", where `#` is computed as 16 * the controller number (starting at zero) + the SCSI address of the unit. Each tape device (of any type) also receives a "`tape#`" alias, where `#` is assigned sequentially across the entire system.

Get Status Request Flavors

The following `device_flavor_t` values apply to SCSI tape devices.

- `MTIOCGET` gets extended status from the drive.
- `TAPE_STATUS` gets the current recording mode and mount information.

Set Status Request Flavors

The following `device_flavor_t` values apply to SCSI tape devices.

- `MTIOCTOP` performs a tape-specific operation.
- `TAPE_STATUS` sets the tape recording mode and mount control flags.

Unsupported Calls

Neither the **device_map** nor the **device_set_filter** calls are supported by the SCSI driver.

APPENDIX B Data Structures

This appendix discusses the specifics of the various structures used as a part of the kernel's various interfaces. This appendix does not discuss all of the various data types used by the kernel's interfaces, only the fields of the various structures used.

host_basic_info

Structure — Defines basic information about a host

SYNOPSIS

```
[1]  struct host_basic_info
[2]  {
[3]          vm_size_t                  max_cpus;
[4]          vm_size_t                  avail_cpus;
[5]          vm_size_t                  memory_size;
[6]          cpu_type_t                 cpu_type;
[7]          cpu_subtype_t              cpu_subtype;
[8]          vm_address_t               min_address;
[9]          vm_address_t               max_address;
[10] };
[11] typedef struct host_basic_info*    host_basic_info_t;
```

DESCRIPTION

The **host_basic_info** structure defines the basic information available about a host.

FIELDS

max_cpus
: Maximum possible CPUs for which kernel is configured

avail_cpus
: Number of CPUs now available

memory_size
: Size of memory, in bytes

cpu_type
: CPU type

cpu_subtype
: CPU sub-type

min_address
: Minimum address in user space

max_address
: Maximum address in user space

NOTES

This structure is machine word length specific because of the memory size returned.

RELATED INFORMATION

Functions: **host_info.**

Data structures: **host_load_info**, **host_sched_info.**

host_load_info

Structure — Defines load information about a host

SYNOPSIS

```
[1] #define CPU_STATE_USER          0
[2] #define CPU_STATE_SYSTEM        1
[3] #define CPU_STATE_IDLE          2
[4] struct host_load_info
[5] {
[6]         load_t                  avenrun[3];
[7]         load_t                  load_factor[3];
[8] };
[9] typedef struct host_load_info*  host_load_info_t;
```

DESCRIPTION

The **host_load_info** structure defines the loading information available about a host. The information returned is exponential averages over three periods of time: 5, 30 and 60 seconds.

FIELDS

avenrun

Load average—average number of runnable processes divided by number of CPUs

load_factor

The processing resources available to a new thread—the number of CPUs divided by (1 + the number of threads)

RELATED INFORMATION

Functions: **host_statistics.**

Data structures: **host_basic_info**, **host_sched_info.**

host_sched_info

Structure — Defines scheduling information about a host

SYNOPSIS

```
[1] struct host_sched_info
[2] {
[3]         mk_rpc_timeout_t            min_timeout;
[4]         mk_rpc_timeout_t            min_quantum;
[5] };
[6] typedef struct host_sched_info*     host_sched_info_t;
```

DESCRIPTION

The **host_sched_info** structure defines the limiting scheduling information available about a host.

FIELDS

min_timeout
: Minimum time-out, in milliseconds

min_quantum
: Minimum quantum (period for which a thread can be scheduled if uninterrupted), in milliseconds

RELATED INFORMATION

Functions: **host_info.**

Data structures: **host_basic_info**, **host_load_info.**

mapped_tvalspec

Structure — Defines format of kernel maintained time in a mapped clock

SYNOPSIS

```
[1] struct mapped_tvalspec
[2] {
[3]         tvalspec_t                  mtv_time;
[4]         unsigned int                mtv_csec;
[5] };
[6] typedef struct mapped_tvalspec      mapped_tvalspec_t;
```

DESCRIPTION

The **mapped_tvalspec** structure defines the format of the current-time structure maintained by the kernel and visible through a mapped clock (**clock_map_time**). The data in this structure is updated at the clock's current resolution and contains the same tvalspec value that would be returned by **clock_get_time**.

FIELDS

mtv_time
: Clock time.

mtv_csec
: A field used to synchronize with the kernel's setting of the time.

NOTES

Because of the race between the referencing of the multiple fields in the clock value and the kernel's setting them, they should be referenced as follows:

```
[1] tvalspec_t*                         ts;
[2] mapped_tvalspec_t*                  mtime;
[3] do
[4] {
[5]         ts → tv_sec = mtime → mtv_time.tv_sec;
[6]         ts → tv_nsec = mtime → mtv_time.tv_nsec;
[7] } while (ts → tv_sec != mtime → mtv_csec);
```

RELATED INFORMATION

Functions: **clock_map_time**, **clock_get_time.**

Data Structures: **tvalspec**.

memory_object_behave_info

Structure — Defines the behavioral attributes for a memory object

SYNOPSIS

```
[1] struct memory_object_behave_info
[2] {
[3]         memory_object_copy_strategy_t   copy_strategy;
[4]         boolean_t                       temporary;
[5] };
[6] typedef struct memory_object_behave_info* memory_object_behave_info_t;
```

DESCRIPTION

The **memory_object_behave_info** structure defines behavior relevant memory object attributes.

FIELDS

copy_strategy

How the kernel should handle copying of regions associated with the memory object. The copy strategy cannot be changed once an object is initialized. Valid values are:

MEMORY_OBJECT_COPY_NONE

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

MEMORY_OBJECT_COPY_DELAY

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

temporary

When the last mapping of the memory object is removed, the object is destroyed without returning any in-memory pages.

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_get_attributes**, **memory_object_change_attributes**, **vm_region**, **vm_set_default_memory_manager**.

Data Structures: **memory_object_perf_info**.

memory_object_perf_info

Structure — Defines the performance attributes for a memory object

SYNOPSIS

```
[1] struct memory_object_perf_info
[2] {
[3]         vm_size_t                       cluster_size;
[4]         boolean_t                       may_cache;
[5] };
[6] typedef struct memory_object_perf_info* memory_object_perf_info_t;
```

DESCRIPTION

The **memory_object_perf_info** structure defines performance relevant memory object attributes.

FIELDS

cluster_size

Preferred cluster size (in bytes) for the memory object. This helps to determine how many pages are transferred in individual data request and return messages.

may_cache

Cache indicator. If true, the kernel can cache data associated with the memory object (keep the memory object active) even if no virtual memory references to it remain.

NOTES

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

RELATED INFORMATION

Functions: **memory_object_establish**, **memory_object_get_attributes**, **memory_object_change_attributes**, **vm_region**, **vm_set_default_memory_manager**.

Data Structures: **memory_object_behave_info**.

mk_rpc_data

Structure — Defines information for RPC services

SYNOPSIS

```
[1]  struct mk_rpc_desc
[2]  {
[3]          mk_rpc_size_t              n_mud; /* Marshalled User Data */
[4]          mk_rpc_short_t             n_reply_ool;
[5]          mk_rpc_short_t             n_ool;
[6]          mk_rpc_size_t              n_ports;
[7]  };
[8]  typedef struct mk_rpc_desc         mk_rpc_desc_t;
[9]
[10] struct mk_rpc_data
[11] {
[12]         mk_rpc_desc_t              descr;
[13]         uchar_t                    payload[sizeof(natural_t)];
[14] }
[15] typedef struct mk_rpc_data         mk_rpc_data_t;
[16]
[17] struct mk_rpc_ool
[18] {
[19]         vm_address_t               address;
[20]         mk_rpc_size_t              size;
[21] };
[22] typedef struct mk_rpc_ool          mk_rpc_ool_t;
[23]
[24] struct mk_rpc_port
[25] {
[26]         mach_port_name_t           portname;
[27] };
[28] typedef struct mk_rpc_port         mk_rpc_port_t;
[29]
[30] union mk_rpc_transit {
[31]         mk_rpc_port_t              port;
[32]         mk_rpc_ool_t               ool;
[33] }
[34] typedef union mk_rpc_transit       mk_rpc_transit_t;
```

DESCRIPTION

The **mk_rpc_data** structure defines the information used by RPC services.

FIELDS

method_ID
Indicates the method to be invoked by the task servicing this RPC.

n_mud
Describes the input data for this RPC; number of bytes of Marshalled User Data.

n_reply_ool
Describes the number of Out-Of-Line (OOL) descriptors supplied by the RPC invoker, indicating regions of memory into which the server's reply data will be deposited. The actual descriptors are supplied at the end of the payload, and the first reply OOL descriptor indicates the buffer which will receive the regular (non-OOL) reply data, formatted as a **mk_rpc_data_t**. See **mk_rpc** for the details of OOL usage.

n_ool
Describes the input data for this RPC; number of OOL descriptors. The descriptors are address/size pairs of type **mk_rpc_ool_t**.

n_ports
Describes the input data for this RPC; number of ports.

payload
Holds the input data for this RPC, consisting of marshalled user data followed by an array of **mk_rpc_transit_t** descriptors for the OOL regions, and ports, in that order, with each section naturally aligned. Lastly, any OOL descriptors which describe the memory which is to receive reply data is included; the first points to the regular (non-OOL) receive buffer, the remainder describe the actual OOL regions.

NOTES

This structure is machine word length sensitive due to the presence of ports and the method_ID.

RELATED INFORMATION

Functions: **mk_rpc, server_dispatch, mk_rpc_reply.**

policy_fifo_info

Structure — Defines first-in-first-out scheduling information

SYNOPSIS

```
[1] struct policy_fifo_base
[2] {
[3]         priority_t                  base_priority;
[4] };
[1] struct policy_fifo_limit
[2] {
[3]         priority_t                  max_priority;
[4] };
[1] struct policy_fifo_info
[2] {
[3]         priority_t                  max_priority;
[4]         priority_t                  base_priority;
[5]         boolean_t                   depressed;
[6] };
```

DESCRIPTION

The **policy_fifo_info** structure defines the first-in-first-out scheduling policy information.

FIFO threads have two priorities associated with them by the system:

- A maximum priority value that can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set.
- A priority value that can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task.

FIELDS

base_priority
: Scheduling priority

max_priority
: Maximum scheduling priority

depressed
: True if scheduling priority is depressed

RELATED INFORMATION

Functions: **thread_info**, **task_info**, **processor_set_info**, **processor_set_policy_control**, **task_policy**, **thread_policy**.

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

policy_rr_info

Structure — Defines round-robin scheduling information

SYNOPSIS

```
[1] struct policy_rr_base
[2] {
[3]         priority_t              base_priority;
[4]         mk_rpc_timeout_t        quantum;
[5] };
[1] struct policy_rr_limit
[2] {
[3]         priority_t              max_priority;
[4] };
[1] struct policy_rr_info
[2] {
[3]         priority_t              max_priority;
[4]         priority_t              base_priority;
[5]         mk_rpc_timeout_t        quantum;
[6]         boolean_t               depressed;
[7] };
```

DESCRIPTION

The **policy_rr_info** structure defines the round-robin scheduling policy information.

Round-robin threads have two priorities associated with them by the system:

- A maximum priority value that can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set.
- A priority value that can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task.

FIELDS

base_priority
: Scheduling priority

quantum
: Scheduling quantum (in milliseconds)

max_priority
: Maximum scheduling priority

depressed
: True if scheduling priority is depressed

RELATED INFORMATION

Functions: **thread_info**, **task_info**, **processor_set_info**, **processor_set_policy_control**, **task_policy**, **thread_policy**.

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

policy_timeshare_info

Structure — Defines timeshare scheduling information

SYNOPSIS

```
[1] struct policy_timeshare_base
[2] {
[3]         priority_t                  base_priority;
[4] };
[1] struct policy_timeshare_limit
[2] {
[3]         priority_t                  max_priority;
[4] };
[1] struct policy_timeshare_info
[2] {
[3]         priority_t                  max_priority;
[4]         priority_t                  base_priority;
[5]         priority_t                  cur_priority;
[6]         boolean_t                   depressed;
[7] };
```

DESCRIPTION

The **policy_timeshare_info** structure defines the timeshare scheduling policy information.

Timeshare threads have three priorities associated with them by the system:

- A priority value that can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task.
- A maximum priority value that can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set.
- A scheduled priority value that is used to make scheduling decisions for the thread. This value is determined on the basis of the user priority value by the scheduling policy (for timesharing, this means adding an increment derived from CPU usage).

FIELDS

max_priority
: Maximum scheduling priority

base_priority
: Base scheduling priority

cur_priority
Current scheduling priority

depressed
True if scheduling priority is depressed

RELATED INFORMATION

Functions: **thread_info**, **task_info**, **processor_set_info**, **processor_set_policy_control**, **task_policy**, **thread_policy**.

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

port_attr_engines

Structure — Defines information for a notification port

SYNOPSIS

```
[1] struct port_attr_engines
[2] {
[3]         int                                 num_engines;
[4] };
[5] typedef struct port_attr_engines            port_attr_engines_t;
```

DESCRIPTION

The **port_attr_engines** structure defines information about a notification port.

FIELDS

num_engines
: The number of notification delivery engines associated with the notification port.

RELATED INFORMATION

Functions: **port_get_attributes, port_set_attributes, port_allocate.**

port_attr_name

Structure — Defines information for a port

SYNOPSIS

```
[1] struct port_attr_name
[2] {
[3]         int                         opaque1;
[4]         int                         opaque2;
[5] };
[6] typedef struct port_attr_name       port_attr_name_t;
```

DESCRIPTION

The **port_attr_name** structure defines information about a port.

FIELDS

opaque1

The name supplied to the server when an RPC is invoked on this port. This defaults to the port name unless overridden.

opaque2

The additional information supplied to the server when an RPC is invoked on this port.

RELATED INFORMATION

Functions: **port_get_attributes, port_set_attributes, port_allocate.**

port_attr_status

Structure — Defines information for a port

SYNOPSIS

```
[1] struct port_attr_status
[2] {
[3]         mach_port_name_t        pool;
[4]         port _type_t            port_type;
[5] };
[6] typedef struct port_attr_status  port_attr_status_t;
```

DESCRIPTION

The **port_attr_status** structure defines information about a port.

FIELDS

pool
: Containing port pool, if port represents a receive right.

port_type
: Type of port, as specified by **port_allocate**, if port represents a receive right.

NOTES

This structure is machine word length sensitive due to the presence of the port pool name.

RELATED INFORMATION

Functions: **port_get_attributes, port_allocate.**

processor_set_basic_info

Structure — Defines the basic information about a processor set.

SYNOPSIS

```
[1] struct processor_set_basic_info
[2] {
[3]         cpu_number_t                    processor_count;
[4]         policy_t                        default_policy;
[5] };
[6] typedef struct processor_set_basic_info*    processor_set_basic_info_t;
```

DESCRIPTION

The **processor_set_basic_info** structure defines the basic information available about a processor set.

FIELDS

processor_count
: Number of processors in this set

default_policy
: Default policy to assign to threads whose otherwise assigned policy is not enabled

RELATED INFORMATION

Functions: **processor_set_info**.

Data Structures: **processor_set_load_info**.

processor_set_load_info

Structure — Defines the scheduling statistics for a processor set.

SYNOPSIS

```
[1] struct processor_set_load_info
[2] {
[3]         mk_rpc_type_number_t            task_count;
[4]         mk_rpc_type_number_t            thread_count;
[5]         load_t                          load_average;
[6]         load_t                          load_factor;
[7] };
[8] typedef struct processor_set_load_info* processor_set_load_info_t;
```

DESCRIPTION

The **processor_set_load_info** structure defines the scheduling statistics maintained for a processor set.

FIELDS

task_count
: Number of tasks currently assigned to this processor set

thread_count
: Number of threads currently assigned to this processor set

load_average
: Average number of runnable processes divided by number of CPUs

load_factor
: The processing resources available to a new thread—the number of CPUs divided by (1 + the number of threads)

RELATED INFORMATION

Functions: **processor_set_statistics**.

Data Structures: **processor_set_basic_info**.

task_basic_info

Structure — Defines basic information for tasks

SYNOPSIS

```
[1]  struct task_basic_info
[2]  {
[3]          suspend_t                  suspend_count;
[4]          vm_size_t                  virtual_size;
[5]          vm_size_t                  resident_size;
[6]          tvalspec_t                 user_time;
[7]          tvalspec_t                 user_idle_time;
[8]          tvalspec_t                 system_time;
[9]          policy_t                   policy;
[10] };
[11] typedef struct task_basic_info*    task_basic_info_t;
```

DESCRIPTION

The **task_basic_info** structure defines the basic information array for tasks. The **task_info** function returns this array for a specified task.

FIELDS

suspend_count
: The current suspend count for the task.

virtual_size
: The number of virtual bytes for the task.

resident_size
: The number of resident bytes for the task

user_time
: The total user run time for terminated threads within the task.

user_idle_time
: The total user run time with scheduling depressed for terminated threads within the task.

system_time
: The total system run time for terminated threads within the task.

policy
: Default scheduling policy to apply to new threads.

NOTES

This structure is machine word length sensitive due to the presence of the virtual address sizes.

RELATED INFORMATION

Functions: **task_info**.

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

task_thread_times_info

Structure — Defines thread execution times information for tasks

SYNOPSIS

```
[1] struct task_thread_times_info
[2] {
[3]         tvalspec_t                          user_time;
[4]         tvalspec_t                          user_idle_time;
[5]         tvalspec_t                          system_time;
[6] };
[7] typedef struct task_thread_times_info*      task_thread_times_info_t;
```

DESCRIPTION

The **task_thread_times_info** structure defines thread execution time statistics for tasks. The **task_info** function returns these times for a specified task. The **thread_info** function returns this information for a specific thread.

FIELDS

user_time
: Total user run time for live threads.

user_idle_time
: The total user run time with scheduling depressed for terminated threads within the task.

system_time
: Total system run time for live threads.

RELATED INFORMATION

Functions: **task_info**.

Data Structures: **task_basic_info, thread_info**.

thread_basic_info

Structure — Defines basic information for threads

SYNOPSIS

```
[1]  struct thread_basic_info
[2]  {
[3]          tvalspec_t                 user_time;
[4]          tvalspec_t                 user_idle_time;
[5]          tvalspec_t                 system_time;
[6]          cpu_usage_t                cpu_usage;
[7]          policy_t                   policy;
[8]          run_state_t                run_state;
[9]          thread_flags_t             flags;
[10]         suspend_t                  suspend_count;
[11]         time_t                     sleep_time;
[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.

user_idle_time
: The total user run time with scheduling depressed for terminated threads within the task.

system_time
: The total system run time for the thread.

cpu_usage
: Scaled CPU usage percentage for the thread.

policy
: Scheduling policy in effect

run_state
: The thread's run state. Possible values are:

  TH_STATE_RUNNING
  : The thread is running normally.

TH_STATE_STOPPED
The thread is stopped.

TH_STATE_WAITING
The thread is waiting normally.

TH_STATE_UNINTERRUPTIBLE
The thread is in an un-interruptible wait state.

TH_STATE_HALTED
The thread is halted at a clean point.

flags
Swap/idle flags for the thread. Possible values are:

TH_FLAGS_SWAPPED
The thread is swapped out.

TH_FLAGS_IDLE
The thread is an idle thread.

suspend_count
The current suspend count for the thread.

sleep_time
The number of seconds that the thread has been sleeping.

RELATED INFORMATION

Functions: **thread_info**.

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

tvalspec

Structure — Defines format of system time values

SYNOPSIS

```
[1] struct tvalspec
[2] {
[3]         unsigned_int            tv_sec;
[4]         clock_res_t             tv_nsec;
[5] };
[6] typedef struct tvalspec         tvalspec_t;
```

DESCRIPTION

The **tvalspec** structure defines the format of the time structure supplied to or returned from the kernel. This definition conforms to the Posix 1003.4 *timespec* definition where the *tv_nsec* structure member is valid if ($0 \leq tv_nsec < 10^9$) and the time period described is (($tv_sec \times 10^9$) + *tv_nsec*) nanoseconds.

FIELDS

tv_sec
: Seconds.

tv_nsec
: Nanoseconds.

RELATED INFORMATION

Functions: **clock_get_time, clock_set_time**, **clock_sleep**, **clock_alarm**, **alarm_reply.**

Data Structures: **mapped_tvalspec**.

vm_region_basic_info

Structure — Defines the attributes of a task's memory region.

SYNOPSIS

```
[1]  struct vm_region_basic_info
[2]  {
[3]          vm_prot_t                  protection;
[4]          vm_prot_t                  max_protection;
[5]          vm_inherit_t               inheritance;
[6]          vm_offset_t                offset;
[7]          vm_behavior_t              behavior;
[8]          boolean_t                  wired;
[9]  };
[10] typedef struct vm_region_basic_info*   vm_region_basic_info_t;
```

DESCRIPTION

The **vm_region_basic_info** structure defines the attributes of a memory region returned by **vm_region**.

FIELDS

protection
: The current protection for the region.

max_protection
: The maximum protection allowed for this region.

inheritance
: The inheritance attribute for the region.

offset
: The region's offset into the memory object. The region begins at this offset.

behavior
: Expected reference pattern for the memory. Valid values are:

  VM_BEHAVIOR_DEFAULT
  : The system's default behavior.

  VM_BEHAVIOR_RANDOM
  : No particular order expected.

  VM_BEHAVIOR_SEQUENTIAL
  : Forward sequential order.

VM_BEHAVIOR_RSEQUNTL
Reverse sequential order.

wired
True if the region is wired.

RELATED INFORMATION

Functions: **vm_region**, **vm_inherit**, **vm_reserve**, **vm_behavior_set**, **vm_protect**.

vm_statistics

Structure — Defines statistics for the kernel's use of virtual memory

SYNOPSIS

```
[1]  struct vm_statistics
[2]  {
[3]          vm_size_t                  free_count;
[4]          vm_size_t                  active_count;
[5]          vm_size_t                  inactive_count;
[6]          vm_size_t                  wire_count;
[7]          vm_size_t                  zero_fill_count;
[8]          vm_size_t                  reactivations;
[9]          vm_size_t                  pageins;
[10]         vm_size_t                  pageouts;
[11]         vm_size_t                  faults;
[12]         vm_size_t                  cow_faults;
[13]         vm_size_t                  lookups;
[14]         vm_size_t                  hits;
[15] };
[16] typedef struct vm_statistics*      vm_statistics_t;
```

DESCRIPTION

The **vm_statistics** structure defines the statistics available on the kernel's use of virtual memory. The statistics record virtual memory usage since the kernel was booted.

For related information for a specific task, see the **task_basic_info** structure.

FIELDS

free_count
: The total number of free pages in the system.

active_count
: The total number of pages currently in use and pageable.

inactive_count
: The number of inactive pages.

wire_count
: The number of pages that are wired in memory and cannot be paged out.

zero_fill_count
: The number of zero-fill pages.

reactivations
The number of reactivated pages.

pageins
The number of requests for pages from a pager (such as the i-node pager).

pageouts
The number of pages that have been paged out.

faults
The number of times the **vm_fault** routine has been called.

cow_faults
The number of copy-on-write faults.

lookups
The number of object cache lookups.

hits
The number of object cache hits.

NOTES

This structure is machine word length specific because of the memory sizes returned.

RELATED INFORMATION

Functions: **task_info**, **host_statistics**, **host_page_size**.

Data Structures: **task_basic_info**.

APPENDIX C Error Return Values

This appendix lists the various kernel return values.

Error Code Format

An error code has the following format:

- system code (6 bits). The **err_get_system** (*err*) macro extracts this field.
- subsystem code (12 bits). The **err_get_sub** (*err*) macro extracts this field.
- error code (14 bits). The **err_get_code** (*err*) macro extracts this field.

The various system codes are:

- *err_kern* —kernel
- *err_us* — user space library
- *err_server*— user space servers
- *err_mk_rpc* — RPC errors
- *err_local* — user defined errors

A typical user error code definition would be:

#define SOMETHING_WRONG *err_local* | **err_sub** (13) | 1

MIG Stub Errors

MIG_ARRAY_TOO_LARGE

User specified array not large enough to hold returned array.

MIG_BAD_ARGUMENTS

RPC receiver found an invalid RPC size, invalid header fields or invalid descriptors. This could only happen if an invalidly formatted RPC (i.e., one that did not pass through a MIG marshalling routine) were passed to a MIG de-multiplexing routine.

MIG_BAD_ID

Bad method ID. This is only returned by MIG de-multiplexing (***subsystem*_server**) routines when the method ID in the supplied RPC is not handled by that routine.

RPC Errors

The **mk_rpc** trap can generate the following return codes (as can all kernel interfaces that are functions):

MK_RPC_TOO_SMALL

The specified *invocation_buffer* was smaller than the minimum size for a mk_rpc_desc_t.

MK_RPC_TOO_LARGE

The specified *invocation_buffer* was larger than the maximum size than the receiver was willing to accept, or the reply buffer was not large enough to handle the reply.

MK_RPC_BAD_REQUEST

The RPC header was inconsistent.

MK_RPC_INVALID_DATA

The address of the supplied invocation buffer was invalid.

MK_RPC_INVALID_BUFF

The address of the reply buffer was invalid.

MK_RPC_INVALID_NAME

The supplied *target_port* was invalid or MACH_PORT_NULL.

MK_RPC_NO_MEMORY

A resource shortage prevented the successful completion of the RPC.

MK_RPC_PORT_DEAD

The *target_port* was destroyed before the RPC invocation reached it.

MK_RPC_MISMATCH

Mismatch between request and reply specifications.

MK_RPC_OOL_MISMATCH

Mismatch of OOL size or number of OOL regions.

MK_RPC_ABORTED

The operation was terminated in the kernel.

Generic Kernel Errors

KERN_SUCCESS

Successful completion.

KERN_INVALID_ARGUMENT

The function requested was not applicable to this type of object.

KERN_INVALID_CAPABILITY

The supplied right is dead, null or not of the proper type.

KERN_INVALID_VALUE

A parameter's value was out of range (or possibly ill-formed). Specific error return values are returned if the parameter's value is properly formed and in range, but not a usable value at this time.

KERN_NO_SPACE

The task's port name space is full.

Port Manipulation Errors

KERN_INVALID_NAME

The port name doesn't denote a right in the task.

KERN_INVALID_RIGHT

The port name denotes a right, but not an appropriate right.

KERN_RESOURCE_SHORTAGE

A system resource could not be allocated to fulfill this request. This failure may be temporary.

Virtual Memory Manipulation Errors

KERN_INVALID_ADDRESS

Specified virtual address is not currently valid.

KERN_INVALID_OBJECT

The external memory manager failed to initialize the memory object.

KERN_MEMORY_ERROR

During a page fault, the memory object indicated that the data could not be returned. This failure may be temporary; future attempts to access this same data may succeed, as defined by the memory object.

KERN_PROTECTION_FAILURE

Specified memory is valid, but does not permit the required forms of access or the protection being requested exceeds that permitted.

KERN_RESOURCE_SHORTAGE

A system resource could not be allocated to fulfill this request. This failure may be temporary.

Miscellaneous Kernel Errors

KERN_ABORTED

The operation was aborted.

KERN_FAILURE

A catch-all error for implementation dependent failures.

KERN_INVALID_CLOCK

The port was not a clock port.

KERN_INVALID_DEVICE

The port was not a device port.

KERN_INVALID_HOST

An argument supplied to assert system privilege was not a host control port.

KERN_INVALID_LEDGER

A ledger was required but not supplied.

KERN_INVALID_MEMORY_CONTROL

The port was not a memory cache control port.

KERN_INVALID_POLICY

The specified scheduling policy is not currently enabled for the processor set.

KERN_INVALID_PROCESSOR_SET

An argument supplied to assert processor set privilege was not a processor set control port.

KERN_INVALID_SECURITY

An argument supplied to assert security privilege was not a host security port.

KERN_INVALID_TASK

A task was required but not supplied.

KERN_INVALID_THREAD

A thread was required but not supplied.

KERN_LOCK_OWNED

A thread using **lock_try** failed because the mutex was owned.

KERN_OBJECT_DESTROYED

A thread was waiting for a synchronizer when the synchronizer was destroyed.

KERN_POLICY_LIMIT

The specified scheduling attributes exceeds the thread's limits.

KERN_TIMED_OUT

The thread's specified time period expired.

Kernel Device Errors

D_SUCCESS

Normal device return. This is the same value as KERN_SUCCESS.

D_ALREADY_OPEN

Exclusive-use device already open.

D_DEVICE_DOWN

Device has been shut down.

D_INVALID_OPERATION

The requested operation or option is invalid for this device.

D_INVALID_RECNUM

Invalid record (block) number.

D_INVALID_SIZE

Invalid I/O size, including request for inband transfer exceeding the limit for inband data.

D_IO_ERROR

Hardware I/O error.

D_NO_SUCH_DEVICE

No device with that name, security ID mismatch, or the device is not operational.

D_OUT_OF_BAND

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

D_READ_ONLY

Data cannot be written to this device.

D_WOULD_BLOCK

Operation would block, but D_NOWAIT set.

APPENDIX D

Interface and Structure Index

Console, Keyboard, and Mouse Drivers 282
Device Concurrency and Covert Storage Channels 281
Device Names 280
Error Code Format 324
Error Return Values 3
Floppy Disk Driver 287
Generic Kernel Errors 328
Interface Descriptions 1
Interface Types 2
Kernel Device Errors 333
MIG Server Routines 4
MIG Stub Errors 325
Miscellaneous Kernel Errors. 331
Network Drivers 286
Parameter Types 3
Port Manipulation Errors 329
RPC Errors 326
SCSI Subsystem 288
Serial Communications (TTY) Drivers. 284
Special Forms 2
VGA Drivers. 285
Virtual Memory Manipulation Errors . . 330
add_partition 244
alarm_cancel 150
alarm_create. 148
alarm_notify. 210
alarm_set 151
alarm_terminate 153
bootstrap_arguments 245
bootstrap_ports. 246
catch_exception_raise 212
catch_exception_raise_state. 212
catch_exception_raise_state_identity . . 212
clock_get_attributes 154
clock_get_time. 156
clock_map_time. 157
clock_set_attributes 266
clock_set_time 268
clock_sleep. 158
cv_broadcast 192
cv_create 193
cv_destroy 194
cv_signal 195
cv_wait. 196
cv_wait_timed 197
device_close. 168
device_get_status. 169
device_map 171
device_open 248
device_packet_complete 215
device_packet_request. 173

device_read 175
device_read_complete. 217
device_read_packet. 177
device_read_request 179
device_set_filter 181
device_set_status. 185
device_write 187
device_write_complete 218
device_write_request 189
do_mach_notify_dead_name 219
end_samples 220
get_time_stamp 160
host_basic_info 292
host_get_boot_info 250
host_get_clock_control 251
host_get_clock_service 161
host_info 163
host_kernel_version 165
host_load_info. 294
host_page_size 42
host_processor_set_priv 253
host_reboot 254
host_sample. 255
host_sched_info 295
host_security_id_list 256
host_statistics 257
ledger_create 258
ledger_terminate 269
lock_acquire 198
lock_create 199
lock_destroy 200
lock_release. 201
lock_try 202
mach_host_self 166
mach_task_self 124
mach_thread_self 92
mapped_tvalspec. 296
memory_object_behave_info 297
memory_object_change_attributes . . 72
memory_object_change_completed 224
memory_object_data_error 74
memory_object_data_request 226
memory_object_data_return 228
memory_object_data_supply 76
memory_object_data_unavailable . . 78
memory_object_data_unlock 230
memory_object_destroy 79
memory_object_establish 80
memory_object_get_attributes 82
memory_object_lock_completed . . 232
memory_object_lock_request 84
memory_object_notify 234
memory_object_perf_info 298
memory_object_rejected 237
memory_object_representative 87
memory_object_revoke 90
memory_object_supply_completed . 239
memory_object_terminate 241
mk_rpc 6
mk_rpc_data. 299
mk_rpc_reply 13
policy_fifo_info 301
policy_rr_info 303
policy_timeshare_info 305
port_allocate. 22
port_attr_engines 307
port_attr_name 308
port_attr_status. 309
port_destroy 24
port_extract_right. 25
port_get_attributes 26
port_get_refs 28
port_insert_right. 30
port_mod_refs 31
port_move 33
port_pool_allocate 38
port_pool_destroy 39
port_request_notification 34
port_set_attributes 36
processor_set_basic_info 310
processor_set_load_info 311
processor_set_policy_control 270
processor_set_statistics 272
processor_set_tasks 273
receive_samples 221
rpc_demultiplex 15
semaphore_create. 203
semaphore_destroy 204
semaphore_read_value. 205
semaphore_signal. 206
semaphore_wait 207
semaphore_wait_timed 208
server_dispatch *17*
*subsystem*_server 19
task_basic_info 312
task_create 125
task_create_security_id 260
task_get_bootstrap_port. 130
task_get_exception_ports. 127
task_get_host_name_port 130
task_get_kernel_port 130
task_get_ledger_port 130
task_get_special_port. 130
task_info. 132

task_resume 134
task_sample 135
task_set_bootstrap_port 138
task_set_exception_ports 136
task_set_host_name_port 138
task_set_kernel_port 138
task_set_ledger_port 138
task_set_policy 274
task_set_special_port 138
task_suspend 140
task_swap_exception_ports 141
task_terminate 144
task_thread_times_info 314
task_threads 145
thread_basic_info 315
thread_create 93
thread_create_empty 94
thread_create_running 96
thread_depress_abort 98
thread_get_exception_ports 99
thread_get_kernel_port 102
thread_get_special_port 102
thread_get_state 104
thread_info 105
thread_notify 107
thread_resume 108
thread_sample 109
thread_set_exception_ports 110
thread_set_kernel_port 112
thread_set_policy 276
thread_set_special_port 112
thread_set_state 113
thread_suspend 114
thread_swap_exception_ports 116
thread_switch 119
thread_terminate 121
thread_wire 262
tvalspec 317
vm_allocate 43
vm_behavior_set 45
vm_copy 47
vm_deallocate 49
vm_inherit 51
vm_map 53
vm_protect 57
vm_read 59
vm_region 61
vm_region_basic_info 318
vm_reserve 63
vm_set_default_memory_manager . 263
vm_set_recycle_limit 264
vm_statistics 320
vm_unreserve 65
vm_unwire 66
vm_wire 67
vm_write 69

APPENDIX E

Object Index

A

abstract backing space vm_set_default_memory_manager 263
abstract memory object memory_cache_notif_server 4
memory_object_establish 80
memory_object_representative 87
memory_object_data_request 226
memory_object_data_return 228
memory_object_data_unlock 230
memory_object_rejected 237
memory_object_terminate 241
alarm alarm_create 148
alarm_cancel 150
alarm_set 151
alarm_terminate 153
alarm reply alarm_notify_server 4
alarm_create 148
alarm_notify 210

B

bootstrap task_create 125
task_get_special_port 130
task_set_special_port 138
bootstrap_ports 246
task_create_security_id 260

C

clock control host_get_clock_control 251
clock_set_attributes. 266
clock_set_time 268
clock name alarm_create 148
clock_get_attributes 154
clock_get_time 156
clock_map_time 157
clock_sleep 158
host_get_clock_service. 161
condition variable cv_broadcast 192
cv_create 193
cv_destroy. 194
cv_signal 195
cv_wait 196
cv_wait_timed. 197

D

default pager add_partition 244
bootstrap_ports 246
device device_close 168
device_get_status 169
device_map 171
device_packet_request 173
device_read 175
device_read_packet. 177
device_read_request 179
device_set_filter 181
device_set_status. 185
device_write 187
device_write_request 189
device_open 248
device master bootstrap_ports 246
device_open 248
device reply device_completion_server 4
device_packet_complete. 215
device_read_complete. 217
device_write_complete 218

E

exception exception_server 4
thread_get_exception_ports 99
thread_set_exception_ports. 110
thread_swap_exception_ports. 116
task_create. 125
task_get_exception_ports 127
task_set_exception_ports 136

task_swap_exception_ports 141
catch_exception_raise 212
catch_exception_raise_state 212
catch_exception_raise_state_identity 212
task_create_security_id 260

F

filter device_set_filter 181
device_packet_complete 215

H

host bootstrap bootstrap_arguments 245
bootstrap_ports 246
host control bootstrap_ports 246
host_get_boot_info 250
host_get_clock_control 251
host_processor_set_priv 253
host_reboot 254
host_sample 255
host_statistics 257
ledger_create 258
thread_wire 262
host name host_page_size 42
memory_object_establish 80
memory_object_representative 87
host_get_clock_service 161
host_info 163
host_kernel_version 165
memory_object_notify 234
host self task_create 125
task_get_special_port 130
task_set_special_port 138
mach_host_self 166
task_create_security_id 260

L

ledger control bootstrap_ports 246
ledger_create 258
ledger_terminate 269
ledger name memory_object_establish 80
task_create 125
task_get_special_port 130
task_set_special_port 138
alarm_create 148
device_open 248
ledger_create 258
task_create_security_id 260

M

memory cache control memory_object_change_attributes 72
memory_object_data_error 74
memory_object_data_supply 76
memory_object_data_unavailable 78
memory_object_destroy 79
memory_object_establish 80
memory_object_get_attributes 82
memory_object_lock_request 84
memory_object_representative 87
memory_object_revoke 90
memory_object_change_completed 224
memory_object_data_request 226
memory_object_data_return 228
memory_object_data_unlock 230
memory_object_lock_completed 232
memory_object_notify 234
memory_object_supply_completed 239
memory_object_terminate 241

memory object representative vm_map 53
vm_region 61
memory_object_establish 80
memory_object_representative 87
memory_object_revoke 90
clock_map_time 157
device_map 171
memory_object_notify 234

mutex cv_wait 196
cv_wait_timed 197
lock_acquire 198
lock_create 199
lock_destroy 200
lock_release 201
lock_try 202

N

notify notify_server 4
port_request_notification 34
do_mach_notify_dead_name 219

P

port mk_rpc 6
rpc_demultiplex 15
server_dispatch *17*

processor set control host_processor_set_priv 253
processor_set_policy_control 270
processor_set_statistics 272

processor_set_tasks 273
task_set_policy 274
thread_set_policy 276
processor set name host_processor_set_priv 253

R

reply mk_rpc_reply 13
memory_object_change_attributes 72
memory_object_data_supply 76
memory_object_lock_request 84
device_packet_request 173
device_read_request 179
device_write_request 189
device_packet_complete 215
device_read_complete. 217
device_write_complete 218
memory_object_change_completed 224
memory_object_lock_completed 232
memory_object_supply_completed 239
rpc mk_rpc 6
mk_rpc_reply 13
rpc_demultiplex 15
server_dispatch *17*
*subsystem*_server. 19
port_allocate 22
port_pool_allocate. 38

S

sample sample_server 4
thread_create 93
thread_sample 109
task_create. 125
task_sample. 135
end_samples 220
receive_samples 221
host_sample. 255
task_create_security_id. 260
security rpc_demultiplex 15
server_dispatch *17*
*subsystem*_server. 19
bootstrap_ports 246
host_security_id_list 256
task_create_security_id. 260
semaphore semaphore_create 203
semaphore_destroy 204
semaphore_read_value 205

semaphore_signal 206
semaphore_wait 207
semaphore_wait_timed 208
server alarm_notify_server 4
device_completion_server 4
exception_server 4
memory_cache_notif_server 4
memory_object_notif_server 4
notify_server 4
sample_server 4
rpc_demultiplex 15
server_dispatch *17*
*subsystem*_server 19
port_allocate 22

T

task port_allocate 22
port_destroy 24
port_extract_right 25
port_get_attributes 26
port_get_refs 28
port_insert_right 30
port_mod_refs 31
port_move 33
port_request_notification 34
port_set_attributes 36
port_pool_allocate 38
port_pool_destroy 39
vm_allocate 43
vm_behavior_set 45
vm_copy 47
vm_deallocate 49
vm_inherit 51
vm_map 53
vm_protect 57
vm_read 59
vm_region 61
vm_reserve 63
vm_unreserve 65
vm_unwire 66
vm_wire 67
vm_write 69
thread_create 93
thread_create_empty 94
thread_create_running 96
task_create 125

	task_get_exception_ports 127
	task_get_special_port 130
	task_info 132
	task_resume. 134
	task_sample. 135
	task_set_exception_ports 136
	task_set_special_port 138
	task_suspend 140
	task_swap_exception_ports 141
	task_terminate. 144
	task_threads. 145
	cv_create 193
	lock_create 199
	semaphore_create 203
	task_create_security_id. 260
	processor_set_tasks. 273
	task_set_policy 274
task self	mach_task_self 124
	task_create. 125
	task_get_special_port 130
	task_set_special_port 138
	catch_exception_raise. 212
	catch_exception_raise_state_identity 212
	task_create_security_id. 260
task special	task_get_special_port 130
	task_set_special_port 138
thread	thread_create. 93
	thread_create_empty. 94
	thread_create_running. 96
	thread_depress_abort 98
	thread_get_exception_ports 99
	thread_get_special_port 102
	thread_get_state 104
	thread_info 105
	thread_notify. 107
	thread_resume. 108
	thread_sample. 109
	thread_set_exception_ports. 110
	thread_set_special_port. 112
	thread_set_state. 113
	thread_suspend 114
	thread_swap_exception_ports. 116
	thread_switch 119
	thread_terminate 121
	task_threads. 145

thread_wire 262
thread_set_policy 276
thread self mach_thread_self 92
thread_get_special_port 102
thread_set_special_port. 112
catch_exception_raise. 212
catch_exception_raise_state_identity 212
thread special thread_get_special_port 102
thread_set_special_port. 112

V

virtual paging bootstrap_ports 246
vm_set_default_memory_manager. 263
vm_set_recycle_limit 264