Kernel API Changes

The following pages are the kernel API modifications relative to the MK5.0 Mach Kernel Interface Specification (dated April 16, 1993). The changes fall into the following categories:

a. General evolution of the microkernel interface (MK6).

b. Collocation for servers, providing a performance enhancement (MK6).

c. Asynchronous device driver model (MK6).

d. Scalability improvements (NMK18).

e. Real-Time extensions (RT2).

It should be noted that the previous asynchronous device driver interfaces are still present, in the API, as the two models are exclusive of each other.

All of the interfaces documented in this section should be viewed as experimental and are subject to change or deletion in future releases.

General MicroKernel Interfaces

mach_port_get_attributes

Function — Returns information about a port

SYNOPSIS

kern_return_t **mach_port_get_attributes**

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

DESCRIPTION

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

PARAMETERS

task

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

name

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

flavor

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

MACH_PORT_DNREQUESTS_SIZE

Returns the number of entries in the dead-name requests table as a signed integer. The data is returned in the integer pointed to by *port_info*. The *port_info_count* argument is set to indicate that *port_info* has a single data item.

MACH_PORT_LIMITS_INFO

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

MACH_PORT_RECEIVE_STATUS

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

port_info

[out structure] Information about the specified port.

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

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

RELATED INFORMATION

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

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

mach_port_set_attributes

Function — Sets attributes of a port

SYNOPSIS

kern_return_t **mach_port_set_attributes**

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

DESCRIPTION

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

PARAMETERS

task

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

name

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

flavor

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

MACH_PORT_DNREQUESTS_SIZE

Expands the number of entries in the dead-name requests table. The value is contained in the signed integer pointed to by *port_info*. The *port_info_count* argument is set to indicate that *port_info* has a single data item.

MACH_PORT_LIMITS_INFO

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

port_info

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

port_info_count

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

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

KERN_NO_SPACE
The requested dead-name requests table size exceeds the maximum allowable size.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_allocate_name**, **mach_port_get_attributes**.

Data Structures: **mach_port_limits**.

mach_port_status

Structure — Defines information for a port

SYNOPSIS

```
[1]  struct mach_port_status
[2]  {
[3]          mach_port_name_t        mps_pset;
[4]          mach_port_seqno_t       mps_seqno;
[5]          mach_port_mscount_t     mps_mscount;
[6]          mach_port_msgcount_t    mps_msgcount;
[7]          mach_port_rights_t      mps_sorights;
[8]          boolean_t               mps_srights;
[9]          boolean_t               mps_nsrequest;
[10]         boolean_t               mps_rt;
[11] };
[12] typedef struct mach_port_status     mach_port_status_t;
```

DESCRIPTION

The **mach_port_status** structure defines information about a port.

FIELDS

mps_pset
Containing port set.

mps_seqno
Current sequence number for the port.

mps_mscount
Make-send count.

mps_msgcount
Number in the queue.

mps_sorights
How many send-once rights.

mps_srights
True if send rights exist.

mps_nsrequest
True if no-senders requested.

mps_rt
True if the port is a real-time port.

NOTES

This structure is machine word-length sensitive due to the presence of the port set name.

RELATED INFORMATION

Functions: **mach_port_get_attributes**, **mach_port_set_attributes**.

task_set_port_space

Function — Set a task's port name space table size

SYNOPSIS

```
kern_return_t task_set_port_space
                (mach_port_t                task,
                 int                        table_entries);
```

DESCRIPTION

The **task_set_port_space** function preallocates the specified number of entries in the specified *task*'s IPC name space.

PARAMETERS

task
: [in send right] The port referencing the task whose port name space is to be set.

table_entries
: [in scalar] The number of entries in the port name space table.

RETURN VALUE

KERN_NO_SPACE
: The requested table size exceeds the maximum allowable table size.

RELATED INFORMATION

Functions: **mach_port_allocate**.

Server Collocation

mach_subsystem_create

Function — Register information about an RPC subsystem

SYNOPSIS

kern_return_t **mach_subsystem_create**

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

DESCRIPTION

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

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

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

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

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

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

A port naming the registered subsystem is returned.

PARAMETERS

target_task
: [in task send right] The task for which the subsystem is registered; normally the calling task, but not necessarily.

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

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

subsys
: [out subsystem send right] The port returned that names the registered subsystem.

RETURN VALUE

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

KERN_INVALID_ARGUMENT
: The *target_task* port is not a send right naming a task, or the subsystem size is not within the range of valid sizes.

KERN_INVALID_TASK
: The target task is dead.

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

RELATED INFORMATION

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

Structures: **user_subsystem**.

mach_port_allocate_subsystem

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

SYNOPSIS

kern_return_t **mach_port_allocate_subsystem**

(task_port_t	*task,*
mach_port_t	*subsys,*
mach_port_t *	*name);*

DESCRIPTION

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

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

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

KERN_SUCCESS
: The right is allocated.

KERN_INVALID_TASK
: The space is null.

KERN_INVALID_TASK
The space is dead.

KERN_RESOURCE_SHORTAGE
Couldn't allocate memory.

KERN_NO_SPACE
No room in space for another right.

RELATED INFORMATION

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

thread_activation_create

Function — Create a thread activation

SYNOPSIS

kern_return_t **thread_activation_create**

(task_port_t	*task,*
mach_port_t	*RPC_port,*
vm_offset_t	*user_stack,*
vm_size_t	*stack_size,*
mach_port_t*	*new_act*);

DESCRIPTION

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

The new thread activation will begin each RPC using the stack pointer specified by *user_stack*.

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

PARAMETERS

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

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

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

stack_size
: [in scalar] The size, in bytes, of the user stack.

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

NOTES

The following calls targeted at a thread activation port may not be called on an empty thread activation (and will return KERN_INVALID_ARGUMENT if they are called with one):

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

mig_stub_routine

Typedef — Defines the calling convention for the MIG-generated server stubs

SYNOPSIS

```
[1]  typedef void (*mig_stub_routine_t) (
[2]            mach_msg_header_t              *InHeadP;
[3]            mach_msg_header_t              *OutHeadP);


[4]  typedef mig_stub_routine_t               mig_impl_routine_t;
[5]  typedef kern_return_t                    (*mig_impl_routine_t)(void);
```

DESCRIPTION

The **mig_stub_routine** type is the definition for MIG-generated server stub routines. These routines: unpack the request message, call the server procedure, and pack the reply message.

FIELDS

InHeadP
Pointer to the header for the request message.

OutHeadP
Pointer to the header for the reply message.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_descriptor**.

routine_arg_descriptor

Structure — Defines the nature of each argument in the routine call

SYNOPSIS

```
[1] struct routine_arg_descriptor
[2] {
[3]         routine_arg_type        type;
[4]         routine_arg_size        size;
[5]         routine_arg_size        count;
[6]         routine_arg_offset      offset;
[7] } *routine_arg_descriptor_t;
```

DESCRIPTION

The **routine_arg_descriptor** structure is the definition for the handling of each argument passed to/from a routine.

FIELDS

type
: The type of argument (Port, Array, etc.).

size
: The element size, in bytes.

count
: The number of elements.

offset
: Offset in the list of routine arguments.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_descriptor, rpc_subsystem**.

routine_descriptor

Structure — Defines the nature of each subsystem routine

SYNOPSIS

```
[1] struct routine_descriptor
[2] {
[3]         mig_impl_routine_t                  impl_routine;
[4]         mig_stub_routine_t                  stub_routine;
[5]         unsigned int                        argc;
[6]         unsigned int                        descr_count;
[7]         struct routine_arg_descriptor       *arg_descr;
[8]         unsigned int                        max_reply_msg;
[9] } *routine_descriptor_t;
```

DESCRIPTION

The **routine_descriptor** structure is the definition of each exported function within a subsystem. The structure contains information on the function location, message disassembly routine, and the nature of the argument signature.

FIELDS

impl_routine
: Server function pointer (code entry point).

stub_routine
: Unmarshalling function pointer.

argc
: Number of argument words.

descr_count
: Number of complex argument descriptors.

arg_descr
: Pointer to the beginning of the argument descriptor array (containing *descr_count* entries).

max_reply_msg
: The maximum size for the reply message.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_arg_descriptor, rpc_subsystem**.

rpc_subsystem

Structure — Defines the set of server routines in a subsystem

SYNOPSIS

```
[1]  struct rpc_subsystem
[2]  {
[3]          struct subsystem                  *subsystem;
[4]          mach_msg_id_t                     start;
[5]          mach_msg_id_t                     end;
[6]          unsigned int                      maxsize;
[7]          vm_address_t                      base_addr;
[8]          struct routine_descriptor         routine[1];
[9]          struct routine_arg_descriptor     arg_descriptor[1];
[10] } *rpc_subsystem_t;


[11] #define RPC_SUBSYSTEM_NULL                ((rpc_subsystem_t) 0)
```

DESCRIPTION

The **rpc_subsystem** structure describes the set of server routines that can be invoked by **mach_rpc** on the ports that are registered with the subsystem. For each routine, the routine number is given, along with the address of the implementation function in the server and a description of the arguments of the routine (i.e. its signature).

This structure definition is only a template for what is really a variable-length structure (generated by MIG for each subsystem). The actual structures do not always have one entry in the *routine* array, and also have a varying number of entries in the *arg_descriptor* array. Each routine has an array of zero or more argument descriptors, one for each complex argument. These arrays are all concatenated together to form the *arg_descriptor* field of the subsystem struct. The *arg_descr* field of each routine entry points to a unique sub-sequence within this catenated array. The goal is to keep everything contiguous.

FIELDS

subsystem
: Reserved for system use.

start
: Minimum routine number.

end
: Maximum routine number (+1).

maxsize
: Maximum size of a message received from **mach_msg**.

base_addr
: The address of this **rpc_subsystem** structure in user space.

routine
: Starting point for the array of routine descriptors. The number of entries is equal to *start-end*.

arg_descriptor
: Starting point for the complex argument descriptors. The number of entries is equal to the sum of the *descr_count* fields for all of the routines.

RELATED INFORMATION

Functions: **mach_subsystem_create**, **mach_port_allocate_subsystem**.

Structures: **routine_arg_descriptor**, **routine_descriptor**.

Device Driver

device_get_status

Function — Return the current device status

LIBRARY

#include <**device/device.h**>
#include <**device/ds_status.h**>

SYNOPSIS

kern_return_t **device_get_status**

(device_port_t	*device,*
device_flavor_t	*flavor,*
dev_status_t	*status,*
mach_msg_type_number_t*	*status_count*);

DESCRIPTION

The **device_get_status** function returns status information pertaining to an open device. The possible values for *flavor* as well as the meaning of the returned status information is device-dependent, except for MDIOCGETL, which is generic.

PARAMETERS

device
: [in device send right] A device port to the device to be interrogated.

flavor
: [in scalar] The type of status information requested. The only device generic flavors are:

MDIOCGETL
Returns the current value of the device's I/O operations limit (i.e. the number of outstanding requests on the device).

DEV_GET_SEMAPHORE
Returns an array of the device's semaphore ports to the requesting user-level task. The array of semaphore ports is returned to the *status* parameter and the number of semaphores returned is determined by the *status_count* parameter. The meaning of the semaphores returned is device-dependent.

In the event that the device does not export any semaphores, **device_get_status** returns D_INVALID_OPERATION.

Note: Semaphores are not imported. Device drivers can only signal kernel-created semaphores on their local node.

status

[out array of natural-sized units] The returned device status.

status_count

[pointer to in/out scalar] On input, the reserved size of *status*; on output, the size of the returned device status (in natural-sized units).

RETURN VALUE

D_DEVICE_DOWN

Device has been shut down

D_INVALID_OPERATION

The device driver does not export any semaphores.

D_NO_SUCH_DEVICE

No device with that name, or the device is not operational.

D_OUT_OF_BAND

Out-of-band condition occurred on device (such as typing control-C)

RELATED INFORMATION

Functions: **device_set_status**.

Structures: *flavor* dependent.

device_read_async

System Trap — Read a sequence of bytes from a device object

SYNOPSIS

```
kern_return_t device_read_async
        (device_port_t          device,
         mach_port_t            queue,
         mach_port_t            request_id,
         dev_mode_t             mode,
         recnum_t               recnum,
         io_buf_len_t           bytes_wanted);
```

DESCRIPTION

The **device_read_async** function enqueues a read operation for a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device-dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done_queue send right] The port returned from **io_done_queue_create**.

request_id
: [in send right] A unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_read_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**, **io_done_queue_terminate**, **io_done_queue_wait**.

device_read_async_inband

System Trap — Read a sequence of bytes "inband" from a device object.

SYNOPSIS

kern_return_t **device_read_async_inband**

(device_port_t	*device,*
mach_port_t	*queue,*
mach_port_t	*request_id,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted)*;

DESCRIPTION

The **device_read_async_inband** function enqueues a read operation for a sequence of bytes from a device object. The meaning of *recnum* as well as the specific operation performed is device-dependent. This call differs from **device_read_async** in that the returned bytes are returned "inband" in the completion IPC message (in io_done_result.qd_inline).

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_queue_create**.

request_id
: [in send right] A unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_read_async_inband returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**, **io_done_queue_terminate**, **io_done_queue_wait**.

device_read_overwrite_async

System Trap— Read a sequence of bytes from a device object into the caller's address space.

SYNOPSIS

kern_return_t **device_read_overwrite_async**

(device_port_t	*device,*
mach_port_t	*queue,*
mach_port_t	*request_id,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_len_t	*bytes_wanted,*
io_buf_ptr_t	*buffer*);

DESCRIPTION

The **device_read_overwrite** system trap enqueues a read operation for a sequence of bytes from a device object to be placed directly into the caller's address space. The meaning of *recnum* as well as the specific operation performed is device-dependent.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from the **io_done_-queue_create** call.

request_id
: [in send right] A unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

bytes_wanted
: [in scalar] Size of data transfer.

buffer

[pointer to in array of bytes] Data buffer to be overwritten.

RETURN VALUE

device_read_overwrite_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_-write_async**, **device_write_async_inband**, **io_done_queue_create**, **io_done_-queue_terminate**, **io_done_queue_wait**.

device_write_async

System Trap — Write a sequence of bytes to a device object.

SYNOPSIS

```
kern_return_t device_write_async
                (device_port_t                      device,
                 mach_port_t                        queue,
                 mach_port_t                        request_id,
                 dev_mode_t                         mode,
                 recnum_t                           recnum,
                 io_buf_ptr_t                       data,
                 io_buf_len_t                       bytes_wanted);
```

DESCRIPTION

The **device_write_async** function enqueues a write operation for a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device-dependent.

PARAMETERS

device
[in device send right] A device port to the device to be read.

queue
[in io_done queue send right] The port returned from **io_done_-queue_create**.

request_id
[in send right] A unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
[in scalar] I/O mode value. Meaningful options are:

D_NOWAIT
Do not wait if data is unavailable.

recnum
[in scalar] Record number to be read.

data
[pointer to in array of bytes] Data bytes to be written.

bytes_wanted
[in scalar] Size of data transfer.

RETURN VALUE

device_write_async returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_-read_overwrite_async**, **device_write_async_inband**, **io_done_queue_create**, **io_done_queue_terminate**, **io_done_queue_wait**.

device_write_async_inband

System Trap — Write a sequence of bytes “inband” to a device object.

SYNOPSIS

kern_return_t **device_write_async_inband**

(device_port_t	*device,*
mach_port_t	*queue,*
mach_port_t	*request_id,*
dev_mode_t	*mode,*
recnum_t	*recnum,*
io_buf_ptr_t	*data,*
io_buf_len_t	*bytes_wanted)*;

DESCRIPTION

The **device_write_async_inband** function enqueues a write operation for a sequence of bytes to a device object. The meaning of *recnum* as well as the specific operation performed is device-dependent. This call differs from **device_write_async** in that the bytes to be written are sent “inband” in the request IPC message.

PARAMETERS

device
: [in device send right] A device port to the device to be read.

queue
: [in io_done queue send right] The port returned from **io_done_-queue_create**.

request_id
: [in send right] A unique request identifier that will be passed back as part of the **io_done_result** structure.

mode
: [in scalar] I/O mode value. Meaningful options are:

  D_NOWAIT
  : Do not wait if data is unavailable.

recnum
: [in scalar] Record number to be read.

data
: [pointer to in array of bytes] Data bytes to be written.

bytes_wanted
[in scalar] Size of data transfer.

RETURN VALUE

device_write_async_inband returns only invalid parameter errors.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **io_done_queue_create**, **io_done_queue_terminate**, **io_done_queue_wait**.

io_done_queue_create

Function — Create an io_done_queue kernel object

SYNOPSIS

kern_return_t **io_done_queue_create**
(host_name_port_t *host,*
mach_port_t* *queue*);

DESCRIPTION

The **io_done_queue_create** function is called to create a new instantiation of the kernel object supporting asynchronous read/write operations on a device.

PARAMETERS

host
[in host-name send right] The name (or control) port for the host on which the io_done_queue should be created.

queue
[out io-done-queue send right] The port referencing the created io_done_queue.

RETURN VALUE

KERN_INVALID_ARGUMENT
The *host* argument is not valid.

KERN_RESOURCE_SHORTAGE
The kernel cannot allocate the queue, or the port referencing the queue, due to insufficient available memory.

KERN_SUCCESS
The io_done_queue was created.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_-read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_terminate**, **io_done_queue_wait**.

io_done_queue_terminate

Function — Terminate an io_done_queue kernel object

SYNOPSIS

kern_return_t **io_done_queue_terminate**
(mach_port_t *queue*);

DESCRIPTION

The **io_done_queue_terminate** function is called to destroy a previous instantiation of the kernel object supporting asynchronous read/write operations on a device.

PARAMETERS

queue
: [in io-done-queue send right] The port referencing the io_done_queue to be destroyed.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_-read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**, **io_done_queue_wait**.

io_done_queue_wait

System Trap— Wait on an io_done_queue kernel object

SYNOPSIS

```
kern_return_t io_done_queue_wait
        (mach_port_t                    queue
         io_done_result_t*              result);
```

DESCRIPTION

The **io_done_queue_wait** function is called to obtain the results of a previously requested asynchronous I/O operation. For each **io_done_queue_wait** invocation, the status of one I/O request is returned. If there are no pending I/O completions, **io_done_queue_wait** blocks in the kernel on the address of the completion queue.

PARAMETERS

queue
: [in io-done-queue send right] The port referencing the io_done_queue to be waited on.

result
: [out structure] The data structure to be filled in with the completion status of the I/O operation.

RETURN VALUE

KERN_INVALID_ADDRESS
: The *result* argument is not a valid address.

KERN_INVALID_ARGUMENT
: The *queue* argument is not a valid io_done_queue.

KERN_SUCCESS
: The io_done_result structure (*result*) contains valid information.

KERN_TERMINATED
: The specified *queue*, or the port referencing it, was destroyed between the thread blocking and the wakeup.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_-read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**, **io_done_queue_terminate**.

Data Structures: **io_done_result**.

io_done_result

Structure — Defines asynchronous I/O completion status

SYNOPSIS

```
[1]  struct io_done_result
[2]  {
[3]  struct control_info
[4]  {
[5]          int                      qd_type;
[6]          mach_port_t              qd_reqid;
[7]          kern_return_t            qd_code;
[8]          io_buf_ptr_t             qd_data;
[9]          int                      qd_count;
[10] } qd_control;
[11] io_buf_ptr_inband_t              qd_inline;
[12] };
[13] typedef struct io_done_result    io_done_result_t;
```

DESCRIPTION

The **io_done_result** structure defines the set of information made available by a device driver upon completion of an outstanding asynchronous I/O request.

FIELDS

qd_type
Identifies the type of I/O operation that was requested. Valid values are:

IO_DONE_READ The operation was a device_read_async or device_-read_async_inband.

IO_DONE_OVERWRITE The operation was a device_read_over-write_async.

IO_DONE_WRITE The operation was a device_write_async or de-vice_write_async_inband.

qd_reqid
An opaque data item used to uniquely identify the I/O request to the task. It is typically supplied by the task as an argument to the asynchronous device request. The kernel transparently returns this data item to the task.

qd_code
The return code that would have been returned by the non-asynchronous variant of the I/O call (i.e. device_read for device_read_async). If

the field is set to KERN_SUCCESS, then the *qd_data* and *qd_count* fields are valid.

qd_data
: If the qd_type field is set to IO_DONE_READ or IO_DONE_OVERWRITE, then this field is a pointer to the data; otherwise, this field's contents is undefined.

qd_count
: The number of bytes read or written.

qd_inline
: Contains the actual data (in-line), pointed to by *qd_data*, used in **device_read_async_inband**. In all other cases, this field is undefined.

RELATED INFORMATION

Functions: **device_read_async**, **device_read_async_inband**, **device_read_overwrite_async**, **device_write_async**, **device_write_async_inband**, **io_done_queue_create**, **io_done_queue_terminate**, **io_done_queue_wait**.

MultiComputer Support

norma_node_self

Function — Return the node index of the current host

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t norma_node_self
                (host_t         host,
                 node_t *       node);
```

DESCRIPTION

The **norma_node_self** function returns the node index of the current host.

PARAMETERS

host
: [in host send-right] Name of the host.

node
: [out scalar] Node index of the host.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **norma_task_create**, **norma_task_clone**.

norma_task_teleport

Function — "Clone" a task on a specified node

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t norma_task_teleport
                (task_port_t                    parent_task,
                 boolean_t                      inherit_memory,
                 node_t                         child_node,
                 task_port_t*                   child_task);
```

DESCRIPTION

The **norma_task_teleport** function "clones" a new task from *parent_task* on the specified *node* and returns the name of the new task in *child_task*. The child task acquires shared parts of the parent's address space (see **vm_inherit**) regardless of the inheritance set for the parent's memory regions, although the inheritance for the child's regions will be set to that of the parent's regions. The child task initially contains no threads. The *parent_task* is then terminated.

By way of comparison, tasks created by the standard **task_create** primitive are created on the same node as the parent.

Other than being created on a different node, the new task has the same properties as if created by **task_create**.

PARAMETERS

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

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

child_node
: [in scalar] The node index of the node on which to create the child.

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

NOTES

This call differs from **norma_task_clone** in that the parent task is terminated as part of the **teleport** call.

This call differs from **norma_task_create** in that the inheritance set for the parent's memory regions is ignored; the child always shares memory with the parent.

This call is intended to support process migration, where the inheritance semantics of **norma_task_create** would break migrated programs that depended upon sharing relationships remaining after migration.

This call is not a true task migration call, in that it does not migrate the port space, threads, and other non-address-space attributes of the task.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **norma_task_clone**, **task_create**, **norma_task_create**.

task_set_child_node

Function — Set the node upon which future child tasks will be created

LIBRARY

Not declared anywhere.

SYNOPSIS

```
kern_return_t task_set_child_node
                (mach_port_t                task,
                 int                        child_node);
```

DESCRIPTION

The **task_set_child_node** function specifies a node upon which child tasks will be created. This call exists only to allow testing with unmodified servers. Server developers should use **norma_task_create** instead.

PARAMETERS

task
: [in task port] The task who's children are to be affected.

node
: [in scalar] The index of the node upon which future children should be created.

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

Functions: **norma_task_create**, **norma_task_clone**.

Real-Time Support

lock_acquire

Function — Acquire access rights to a lock

SYNOPSIS

kern_return_t **lock_acquire**
    (lock_set_port_t   *lock_set,*
    int   *lock_id*);

DESCRIPTION

The **lock_acquire** function acquires access rights to a specific lock being represented by a given lock set. If the lock is already controlled by another thread then the calling thread will block.

PARAMETERS

lock_set
[in send right] The port naming the lock set which represents the lock.

lock_id
[in scalar] The lock, represented by the lock set, to be acquired.

RETURN VALUE

KERN_SUCCESS
The lock was acquired.

KERN_INVALID_ARGUMENT
The specified *lock_set* is invalid, or the *lock_id* is out of range.

KERN_LOCK_UNSTABLE
The acquired lock has an unstable state.

KERN_LOCK_SET_DESTROYED
The specified lock has been destroyed.

RELATED INFORMATION

Functions: **lock_release**, **lock_try**, **lock_handoff**, **lock_handoff_accept**, **lock_make_stable**, **lock_set_create**, **lock_set_destroy**.

lock_handoff

Function — Hand-off ownership of a lock

SYNOPSIS

```
kern_return_t lock_handoff
                (lock_set_port_t                    lock_set,
                 int                                lock_id);
```

DESCRIPTION

The **lock_handoff** function hands-off lock ownership from the calling thread to an anonymous accepting thread. The lock must be owned by the calling thread. If the accepting thread is not waiting to receive the lock, the calling thread will block until the hand-off is accepted.

PARAMETERS

lock_set
: [in send right] The port naming the lock set which represents the lock.

lock_id
: [in scalar] The lock, represented by the lock set, to be handed off.

RETURN VALUE

KERN_INVALID_ARGUMENT
: The specified *lock_set* is invalid, or the *lock_id* is out of range.

KERN_INVALID_RIGHT
: The calling thread does not own the lock being handed off.

KERN_SUCCESS
: The lock hand-off was successful.

KERN_LOCK_SET_DESTROYED
: The specified lock has been destroyed.

RELATED INFORMATION

Functions: **lock_acquire**, **lock_release**, **lock_try**, **lock_handoff_accept**, **lock_make_stable**, **lock_set_create**, **lock_set_destroy.**

lock_handoff_accept

Function — Accept a lock hand-off

SYNOPSIS

```
kern_return_t lock_handoff_accept
                (lock_set_port_t                    lock_set,
                 int                                lock_id);
```

DESCRIPTION

The **lock_handoff_accept** function accepts a lock hand-off from an anonymous sending thread. If the sending thread is not waiting to hand-off the lock, the calling thread will block until the lock handoff is completed.

Only one thread may be waiting to accept a lock handoff at any given time.

PARAMETERS

lock_set
: [in send right] The port naming the lock set which represents the lock.

lock_id
: [in scalar] The lock, represented by the lock set, that is the object of the handoff operation.

RETURN VALUE

KERN_ALREADY_WAITING
: Another thread is already waiting for a hand-off of this lock.

KERN_INVALID_ARGUMENT
: The specified *lock_set* is invalid, or the *lock_id* is out of range.

KERN_SUCCESS
: The lock hand-off was successful.

KERN_LOCK_UNSTABLE
: The acquired lock has an unstable state.

KERN_LOCK_SET_DESTROYED
: The specified lock has been destroyed.

RELATED INFORMATION

Functions: **lock_acquire**, **lock_release**, **lock_try**, **lock_handoff**, **lock_make_stable**, **lock_set_create**, **lock_set_destroy**.

lock_make_stable

Function — Clear a lock's unstable state

SYNOPSIS

kern_return_t **lock_make_stable**
(lock_set_port_t *lock_set,*
int *lock_id*);

DESCRIPTION

The **lock_make_stable** function clears the specified lock's unstable state, making the lock's state stable again.

PARAMETERS

lock_set
[in send right] The port naming the lock set which represents the lock.

lock_id
[in scalar] The lock, represented by the lock set, to be stabilized.

RETURN VALUE

KERN_INVALID_ARGUMENT
The specified *lock_set* is invalid, or the *lock_id* is out of range.

KERN_SUCCESS
The lock's state was stabilized.

KERN_LOCK_SET_DESTROYED
The specified lock has been destroyed.

RELATED INFORMATION

Functions: **lock_acquire**, **lock_release**, **lock_try**, **lock_handoff**, **lock_handoff_accept**, **lock_set_create**, **lock_set_destroy**.

lock_release

Function — Release ownership of a lock

SYNOPSIS

kern_return_t **lock_release**
(lock_set_port_t *lock_set,*
int *lock_id*);

DESCRIPTION

The **lock_release** function release the ownership of the specified lock. If the calling thread does not own the lock then the call will fail.

PARAMETERS

lock_set
[in send right] The port naming the lock set which represents the lock.

lock_id
[in scalar] The lock, represented by the lock set, to be released.

RETURN VALUE

KERN_INVALID_ARGUMENT
The specified *lock_set* is invalid, or the *lock_id* is out of range.

KERN_SUCCESS
The ownership of the lock was released.

KERN_FAILURE
The calling thread does not own the lock.

KERN_LOCK_SET_DESTROYED
The specified lock has been destroyed.

RELATED INFORMATION

Functions: **lock_acquire**, **lock_make_stable**, **lock_try**, **lock_handoff**, **lock_handoff_accept**, **lock_set_create**, **lock_set_destroy**.

lock_try

Function — Attempt to acquire access rights to a lock

SYNOPSIS

kern_return_t **lock_try**
(lock_set_port_t *lock_set,*
int *lock_id*);

DESCRIPTION

The **lock_try** function attempts to acquire the specified lock without blocking. The return value indicates whether the lock was acquired.

PARAMETERS

lock_set
[in send right] The port naming the lock set which represents the lock.

lock_id
[in scalar] The lock, represented by the lock set, to be acquired.

RETURN VALUE

KERN_INVALID_ARGUMENT
The specified *lock_set* is invalid, or the *lock_id* is out of range.

KERN_SUCCESS
The lock was acquired.

KERN_LOCK_UNSTABLE
The acquired lock has an unstable state.

KERN_LOCK_SET_DESTROYED
The specified lock has been destroyed.

KERN_LOCK_OWNED
Another thread currently owns the requested lock.

KERN_LOCK_OWNED_SELF
The calling thread already owns the lock.

RELATED INFORMATION

Functions: **lock_acquire**, **lock_make_stable**, **lock_release**, **lock_handoff**, **lock_handoff_accept**, **lock_set_create**, **lock_set_destroy**.

lock_set_create

Function — Create a new lock set

LIBRARY

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

SYNOPSIS

kern_return_t **lock_set_create**

(task_port_t	*task*,
lock_set_port_t*	*lock_set,*
int	*locks*
int	*policy*);

DESCRIPTION

The **lock_set_create** function creates a new lock set representing a collection of associated locks. The lock set is associated with the specified task. A send right naming the lock set is returned to the caller.

PARAMETERS

task
The task receiving the send right to the newly created lock set.

lock_set
[out send right] The port naming the lock set which represents the lock.

locks
[in scalar] The number of locks the lock set will represent (must be a positive value).

policy
[in scalar] The blocked thread wakeup policy for the newly created lock set. Valid policies are:

SYNC_POLICY_FIFO
a first-in-first-out policy for scheduling thread wakeup.

SYNC_POLICY_FIXED_PRIORITY
a fixed priority policy for scheduling thread wakeup.

RETURN VALUE

KERN_SUCCESS
The lock set was created.

KERN_INVALID_ARGUMENT
Either the *task* or *policy* argument is invalid, or the *locks* argument is a negative value.

KERN_RESOURCE_SHORTAGE
The kernel could not allocate the lock set.

RELATED INFORMATION

Functions: **lock_acquire**, **lock_make_stable**, **lock_try**, **lock_handoff**, **lock_handoff_accept**, **lock_try**, **lock_set_destroy**.

lock_set_destroy

Function — Destroy a lock set and its associated locks

SYNOPSIS

```
kern_return_t lock_set_destroy
                (task_port_t                task,
                 lock_set_port_t            lock_set);
```

DESCRIPTION

The **lock_set_destroy** function will destroy a lock set and all of its associated locks. Threads that are blocked on locks represented by the destroyed lock set are unblocked and will receive a KERN_LOCK_SET_DESTROYED error message indicating that the lock set was destroyed. The **lock_set_destroy** function will only succeed if the specified task is associated with the specified lock set.

PARAMETERS

task
: The task associated with the lock set.

lock_set
: [in send right] The port naming the lock set being destroyed.

RETURN VALUE

KERN_INVALID_ARGUMENT
: The specified *lock_set* is invalid.

KERN_SUCCESS
: The lock set was destroyed.

RELATED INFORMATION

Functions: **lock_acquire**, **lock_make_stable**, **lock_try**, **lock_handoff**, **lock_handoff_accept**, **lock_try**, **lock_set_create**.

semaphore_create

Function — Create a new semaphore

LIBRARY

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

SYNOPSIS

```
kern_return_t semaphore_create
        (task_port_t            task,
         semaphore_port_t       *semaphore,
         int                    policy,
         int                    value);
```

DESCRIPTION

The **semaphore_create** function creates a new semaphore, associates the created semaphore with the specified task, and returns a send right naming the new semaphore.

In order to support a robust producer/consumer communication service, Interrupt Service Routines (ISR) must be able to signal semaphores. The semaphore synchronizer service is designed to allow user-level device drivers to perform signal operations, eliminating the need for event counters.

Device drivers which utilize semaphores are responsible for creating (via **semaphore_create**) and exporting (via **device_get_status**) semaphores for user level access. Device driver semaphore creation is done at device initialization time. Device drivers may support multiple semaphores.

PARAMETERS

task

[in task port] The task receiving the send right of the newly created semaphore.

semaphore

[out send right] The port naming the created semaphore.

policy

[in scalar] The blocked thread wakeup policy for the newly created semaphore. Valid policies are:

SYNC_POLICY_FIFO
a first-in-first-out policy for scheduling thread wakeup.

SYNC_POLICY_FIXED_PRIORITY
a fixed priority policy for scheduling thread wakeup.

value
[in scalar] The initial value of the semaphore count.

RETURN VALUE

KERN_INVALID_ARGUMENT
The *task* argument or the *policy* argument was invalid.

KERN_RESOURCE_SHORTAGE
The kernel could not allocate the semaphore.

KERN_SUCCESS
The semaphore was successfully created.

RELATED INFORMATION

Functions: **semaphore_destroy**, **semaphore_signal**, **semaphore_signal_all**, **semaphore_wait**, **device_get_status**.

semaphore_destroy

Function — Destroy a semaphore

SYNOPSIS

kern_return_t **semaphore_destroy**
(task_port_t *task,*
semaphore_port_t *semaphore*);

DESCRIPTION

The **semaphore_destory** function destroys a semaphore. All send rights naming the semaphore become dead names. Threads waiting on the semaphore become unblocked with the return from the **semaphore_wait** call indicating that the semaphore was destroyed. The **semaphore_destory** will only succeed if the semaphore is associated with the specified task.

PARAMETERS

task
[in task port] The task associated with the target semaphore.

semaphore
[in send right] The port naming the semaphore to be destroyed.

RETURN VALUE

KERN_INVALID_ARGUMENT
Either, or both, the *task* or *semaphore* arguments were invalid.

KERN_INVALID_RIGHT
The specified *task* does not own the specified *semaphore*.

KERN_SEMAPHORE_DESTROYED
The specified *semaphore* was previously destroyed.

KERN_SUCCESS
The semaphore was destroyed.

RELATED INFORMATION

Functions: **semaphore_create**, **semaphore_signal**, **semaphore_signal_all**, **semaphore_wait**, **device_get_status**.

semaphore_signal

Function — Increments the semaphore count

SYNOPSIS

kern_return_t **semaphore_signal**
(semaphore_port_t *semaphore*);

DESCRIPTION

The **semaphore_signal** function increments the semaphore count. If the count goes non-negative (i.e. greater than or equal to 0) and a thread is blocked on the semaphore, then the waiting thread is scheduled to execute. If multiple threads are blocked on the semaphore, the thread scheduled to execute is selected according to the wakeup policy of the semaphore (set when the semaphore was created via **semaphore_create**).

Device driver interrupt service routines may safely execute **semaphore_signal** operations without causing a deadlock.

PARAMETERS

semaphore
[in send right] The port naming the semaphore to be signalled.

RETURN VALUE

KERN_INVALID_ARGUMENT
The specified semaphore is invalid.

KERN_SEMAPHORE_DESTROYED
The specified semaphore has been destroyed.

KERN_SUCCESS
The semaphore has been signalled.

RELATED INFORMATION

Functions: **semaphore_create**, **semaphore_destroy**, **semaphore_signal_all**, **semaphore_wait**, **device_get_status**.

semaphore_signal_all

Function — Wakeup all threads blocked on a semaphore

SYNOPSIS

kern_return_t **semaphore_signal_all**
(semaphore_port_t *semaphore*);

DESCRIPTION

The **semaphore_signal_all** function wakes up all of the threads blocked on the semaphore. The semaphore count is reset to zero.

PARAMETERS

semaphore
[in send right] The port naming the semaphore to be signalled.

RETURN VALUE

KERN_INVALID_ARGUMENT
The specified semaphore is invalid.

KERN_SEMAPHORE_DESTROYED
The specified semaphore has been destroyed.

KERN_SUCCESS
The semaphore has been signalled.

RELATED INFORMATION

Functions: **semaphore_create**, **semaphore_destroy**, **semaphore_signal**, **semaphore_wait**, **device_get_status**.

semaphore_wait

Function — Decrement the semaphore count

SYNOPSIS

```
kern_return_t semaphore_wait
        (semaphore_port_t                    semaphore);
```

DESCRIPTION

The **semaphore_wait** function decrements the semaphore count. If the semaphore count is negative after decrementing, the calling thread blocks.

Device driver interrupt service routines (ISR) should never execute **semaphore_wait**, since waiting on a semaphore at the ISR level may, and often will, lead to a deadlock.

PARAMETERS

semaphore
: [in send right] The port naming the semaphore that the wait operation is being performed upon.

RETURN VALUE

KERN_INVALID_ARGUMENT
: The specified semaphore is invalid.

KERN_SEMAPHORE_DESTROYED
: The specified semaphore has been destroyed.

KERN_SUCCESS
: The semaphore wait operation was successful.

RELATED INFORMATION

Functions: **semaphore_create**, **semaphore_destroy**, **semaphore_signal**, **semaphore_signal_all**, **device_get_status**.

mach_rpc_trap

System Trap— Real-Time RPC trap

LIBRARY

#include <mach/rpc.h>

SYNOPSIS

mach_rpc_return_t **mach_rpc_trap**
    (mach_port_t *dest_port,*
    mach_rpc_id_t *routine_num*,
    mach_rpc_signature_t *signature_ptr*,
    mach_rpc_size_t *signature_size*);

DESCRIPTION

The **mach_rpc_trap** is accessed via the MIG generated **MACH_RPC** macro which is invoked transparently when the **mach_rpc** feature is enabled.

PARAMETERS

dest_port
: [in send right] The port representing the destination of the RPC (usually a registered subsystem, see **mach_port_allocate_subsystem**).

routine_num
: [in scalar] Identifier of the server work function.

signature_ptr
: [in pointer] Pointer to the call's **mach_rpc_signature** structure.

signature_size
: [in scalar] Size, in bytes, of the call's **mach_rpc_signature** structure.

RETURN VALUE

KERN_FAILURE
: Either the argument copyin failed, there were too many arguments, or the argument copyout failed.

KERN_INVALID_ARGUMENT
: The *dest_port*, *signature_ptr*, and/or the subsystem associated with the *dest_port* is invalid; the *siganture_size* is less than, or equal to, zero.

KERN_NO_ACCESS
: The kernel port associated with the *dest_port* name is a norma proxy port.

KERN_RESOURCE_SHORTAGE
: The kernel could not allocate storage for an internal rpc state structure.

NOTES

This function is not designed for use directly by the user. It is automatically generated by MIG to handle a function call. For a complete description of this functionality, refer to: Burke, Edward, Michael Condict, David Mitchell, Franklin Reynolds, Peter Watkins, William Willcox, “RPC Design for Real-Time Mach”, OSF Research Institute, Cambridge, MA, <date>.

RELATED INFORMATION

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

Structures: **mach_rpc_signature, routine_arg_descriptor**, **routine_descriptor**, **rpc_subsystem**.

mach_rpc_return_trap

System Trap— Real-Time RPC trap return location

LIBRARY

#include <mach/rpc.h>

SYNOPSIS

kern_return_t **mach_rpc_return_trap**
( );

DESCRIPTION

The **mach_rpc_return_trap** is used as the return address for the server work function (or MIG generated function) invoked via the **mach_rpc_trap** call.

PARAMETERS

NONE.

RETURN VALUE

The return value is specific to the server function actually executed via the **mach_rpc_trap** call.

NOTES

For a complete description of this functionality, refer to: Burke, Edward, Michael Condict, David Mitchell, Franklin Reynolds, Peter Watkins, William Willcox, "RPC Design for Real-Time Mach", OSF Research Institute, Cambridge, MA, <date>.

RELATED INFORMATION

Functions: **mach_rpc_trap**.

Structures: **mach_rpc_signature**, **routine_arg_descriptor**, **routine_descriptor**, **rpc_subsystem**.

mach_rpc_signature

Structure— Signature for the Mach RPC paradigm

SYNOPSIS

```
[1]  struct mach_rpc_signature
[2]  {
[3]          struct routine_descriptor          rd;
[4]          struct routine_arg_descriptor      rad[1];
[5]  } *mach_rpc_signature_t;
```

DESCRIPTION

The **mach_rpc_signature** structure is a template. MIG automatically generates the appropriate values as part of creating the actual **mach_rpc** trap.

FIELDS

rd
The **routine_descriptor** structure for this call.

rad
The **routine_arg_descriptor** structure for this call.

RELATED INFORMATION

Functions: **thread_activation_create**, **mach_port_allocate_subsystem**, **mach_rpc, mach_subsystem_create**.

Structures: **routine_arg_descriptor**, **routine_descriptor**.