OSF–RI MK6.1
Release Notes

By
Marc J. Teller

Revision 1.3

OSF RESEARCH INSTITUTE
1 Cambridge Center
Cambridge, MA 02142

Feature Description **1**
Introduction.. 1
Collocated Servers 2
Asynchronous Driver Model 2
MultiComputer Support 3
Real-Time Extensions 3
Microkernel Performance Suite 5
Reference Papers 5

Installation Notes **7**
Introduction.. 7
Reference Platform Definition 7
Installing The Source... 8
Building The Source 8
1. Building The Release 9
2. Booting The Built Images 14
Common Problems... 15
Debugging {with} OSF/1 MK 15
OSF/1 MK Paging 16

MK6.1 Known Problems **18**
Introduction.. 18
Unknown 18
Kernel 19
Server... 20
Commands 23

Kernel API Changes **25**
General MicroKernel Interfaces
mach_port_get_attributes..27
mach_port_set_attributes..29
mach_port_status 31
task_set_port_space 33
Server Collocation
mach_subsystem_create...35
mach_port_allocate_subsystem 37
thread_activation_create 39
mig_stub_routine 41
routine_arg_descriptor 42
routine_descriptor 43
rpc_subsystem..44
Device Driver
device_get_status 47

device_read_async ... 49
device_read_async_inband ... 51
device_read_overwrite_async ... 53
device_write_async ... 55
device_write_async_inband ... 57
io_done_queue_create ... 59
io_done_queue_terminate ... 60
io_done_queue_wait ... 61
io_done_result ... 63
MultiComputer Support
norma_node_self ... 66
norma_task_teleport ... 67
task_set_child_node ... 69
Real-Time Support
lock_acquire ... 71
lock_handoff ... 72
lock_handoff_accept ... 73
lock_make_stable ... 75
lock_release ... 76
lock_try ... 77
lock_set_create ... 78
lock_set_destroy ... 80
semaphore_create ... 81
semaphore_destroy ... 83
semaphore_signal ... 84
semaphore_signal_all ... 85
semaphore_wait ... 86
mach_rpc_trap ... 87
mach_rpc_return_trap ... 89
mach_rpc_signature ... 90

Feature Description

Introduction

The MK6.1 release is comprised of results from several projects at the Open Software Foundation – Research Institute. These projects are:

a. General evolution of the microkernel interface.

b. The MK6 performance improvement project, which has contributed the server collocation component of the release.

c. The MK6 device driver asynchronous I/O model, which allows task notification directly from interrupt context.

d. The NMK18 NoRMA project,which has contributed enhancements to the multicomputer support component of the release.

e. The RT2 real-time project, which has contributed fundamental extensions to the kernel in support of real-time applications and environments.

f. The Microkernel Performance Suite.

Each of these projects, except for the general evolution of the interface, is described in greater detail in the following sections.

Collocated Servers

Most of the performance penalties associated with Microkernel (MK) systems result from locating an OS server in user space and using a general-purpose IPC message mechanism to implement RPC, thus adding overhead to the system call path. Therefore the principal strategy for improving MK performance has been to move the OS server task back into the kernel address space and to optimize the RPC mechanism for the case where the client and server share an address space.

Collocation means locating separately-compiled and linked programs in the same address space. In the current implementation of collocation, each program runs in a separate task. This corresponds to the original model of collocating several tasks in the same address space. However, the model is evolving in the direction of having several separately compiled and linked programs loaded into a single task.

When programs share the same address space, it becomes possible to replace IPC message-based RPC with a form of direct procedure call that is called *short-circuited RPC*. This is many times more efficient than message passing, and accounts for the largest single performance improvement. One of the requirements for this project was to maintain the modularity of the MK design. In the case of RPC, this means that collocated programs must not be aware that they are communicating via short-circuited rather than message-based RPC.

More information can be found in the paper [5] included in the release package.

Asynchronous Driver Model

In the current Microkernel (MK) asynchronous I/O completion notifications are posted to a task using a reply message generated by the MK. Because messages cannot be sent from interrupt context, an `io_done` thread lives in the MK to perform the messaging function. This `io_done` thread introduces an additional thread context switch, with the accompanying performance penalty, on each asynchronous I/O completion. A new mechanism has been introduced in the MK by which any server can circumvent the interposition of the `io_done` thread on each asynchronous I/O completion.

Instead of waiting on a port set for I/O completion requests, a task waits on a new kernel object called an `io_done_queue`. Applications such as the UNIX single server create a single `io_done_queue` at server initialization time. The corresponding wait operation returns the I/O completion status. For each wait invocation, the status of one I/O request is returned. If there are no pending I/O completions the wait operation blocks in the kernel on the address of the completion queue. The MK, from interrupt context, enqueues (FIFO) completions on the completion queue and posts a wakeup on it for each I/O completion. Completion processing previously done by the MK `io_done` thread is now done by the task thread when it awakens.

MultiComputer Support

This release updates the NoRMA support in Mach 3.0 to work with the current kernel source base, and makes NoRMA more scalable by adding MP safety and efficiency. Thus NoRMA now uses untyped IPC, NoRMA itself is MP-safe (i.e., has appropriate lock/unlock protocols for use on multiprocessors), and the underlying XMM implementation has been made MP-efficient. Much work has gone into increased robustness and into completing the task of making all kernel VM services work transparently in the distributed case.

Real-Time Extensions

This release is a continuation of the effort by the OSF-RI to provide support for real-time applications and environments. The major areas of real-time functionality provided in MK6.1 are:

- *Synchronizers*

OSF MK6.1 supports two new synchronizers: semaphores and locks (see [4] for more information).

Semaphores are a simple and powerful synchronization primitive that can be used to implement a host of other synchronization protocols. The semaphore abstraction is particularly well-suited for inter-task producer/-consumer style communication. Tasks may dynamically create and destroy semaphores as needed; additional operations include signalling, and waiting on, a semaphore. Semaphores are named by port rights, enabling operations on semaphores to be network-transparent.

Locks are first class entities, which can be created and destroyed, as well as locked and unlocked. Lock operations are performed against *lock sets*. A lock set represents either a single lock or a collection of associated locks. Lock sets are named by ports, to allow them to be referenced across task boundaries; while they are owned by tasks, the locks are owned by threads. The thread that acquires a lock becomes the owner of the lock. A lock can be released only by the thread who owns the lock. Lock ownership may be *handed-off* between threads.

- *Real-Time threads*

  This is supplied by a user-level library and is not, therefore, documented in the Kernel API Changes chapter of the Release Notes. See [3] for a complete description of the library functionality.

- *RPC's for Real-Time*

  Recent work on systems that employ message-based IPC has demonstrated that optimizing the Remote Procedure Call (RPC) operation yields significant performance improvements. One of the goals of the OSF/RI MK project is to realize these improvements in the current development path for the real-time Mach kernel. This work leverages the collocation effort and the redefinition of a thread into two components: a shuttle and an activation.

Perhaps the most important difference between RT-RPC and Mach IPC based RPC is the treatment of message meta data. The RPC parameter data, and the IN and OUT arguments to the remote service, will be treated separately from the meta data. Every effort will be made to use the parameter data, without additional copies, as it is provided by the standard 'C' compiler of each target platform. This contrasts with the **mach_msg** approach which requires the parameter data to be coped to the message body. It is possible to send a message or an RPC to the same port. For a more detailed description of this functionality, see the included paper [2].

In order to better present the changes to the API, it is important to briefly define a number of terms:

- **Thread** – Normally an execution stack and processor state. In Mach, and other systems, a thread is also a schedulable object that contains scheduling parameters and, possibly scheduling policy information.
- **Activation** – A Mach thread that is currently associated with a shuttle.
- **Unbound Activation** – An activation that is not currently associated with a shuttle.
- **Shuttle** – This contains the scheduling policy and parameters, as well as cpu usage and a pointer to the current thread. The shuttle is the part of the thread/shuttle system that can migrate, while the thread part remains with the task in which it was created.
- **RPC chain** – Active threads (i.e. activations) in different tasks linked by RPC calls.
- **Alert** – A software generated exception that affects the thread at which the shuttle is currently pointing.
- **Signature** – An array of descriptors that describe the kernel visible arguments (e.g. ports) pushed on the stack for an RPC call.
- **Subsystem** – A data structure registered by an RPC target task to allow the kernel to marshall and unmarshall arguments and directly invoke task work functions.

Microkernel Performance Suite

Most OSF/1 MK performance measurements are accomplished by running typical Unix programs on top of the server which characterizes the overall system performance. For system developers, the performance figures obtained with such methods are not sufficient to analyze the performance of the microkernel as an individual component.

The Microkernel Performance Suite (MKPS) is a framework to take multiple measurements of each system service and compute the average performance and its standard deviation. It calibrates the number of iterations to get a significant measure (it has to perform enough iterations to get a low variation in the measure, but not too many to make sure that the results are available after a reasonable time), and automatically invokes the tests selected by the user. It also provides some tools to detect resources leak that might affect the results. The set of microkernel services measured involve IPC, VM, exception handling, scheduling, port, task and thread manipulation, EMMI, and device services. Since there are no kernel API modifications for this feature, see [1] for a complete description.

Reference Papers

[1] Barbou des Places, François, Philippe Bernadat,"OSF Microkernel Benchmark Suite," OSF Research Institute, Cambridge, MA, July, 1994.

[2] Burke, Edward, Michael Condict, David Mitchell, Franklin Reynolds, Peter Watkins, William Willcox, "RPC Design for Real-Time Mach," OSF Research Institute, Cambridge, MA, April, 1994.

[3] CaraDonna, Joseph, "OSF–RI MK6.1 Real-Time Threads Library: Rthreads Interface Guide," OSF Research Institute, Cambridge, MA, *DRAFT*: September 8, 1994.

[4] CaraDonna, Joseph, Robert Haydt, Franklin Reynolds, "New Synchronization Services for OSF Mach: Synchronizers," OSF Research Institute, Cambridge, MA, April 12, 1994.

[5] Condict, Michael, Don Bolinger, Eamonn McManus, David Mitchell, Steve Lewontin, "Microkernel Modularity With Integrated Kernel Performance,",OSF Research Institute, Cambridge, MA, April, 1994.

[6] Haydt, Robert, Joseph CaraDonna, Franklin Reynolds, "Mach Scheduling Framework," OSF Research Institute, Cambridge, MA, April, 1994.

[7] Langerman, Alan, David Black, Michelle Dominijanni, Steve Sears, Randall Dean, Dejan Milojicic, "NORMA IPC Version Two: Architecture and Design," OSF Research Institute, Cambridge, MA, April, 1994.

Installation Notes

Introduction

This chapter describes the installation procedures for the supported reference platform. The installation kit in this snapshot includes only the sources needed to build the MK6 OSF/1 server and microkernel, together with supporting Mach-related software. This software must be built on a machine already installed with OSF/1 R1.2 or later.

There is a mailing list (`mk6-request@osf.org`) for announcement of new releases, bug fixes and for discussion among collaborators. Send mail to `mk6-request-request@osf.org` if you would like to be added to this list or if you experience problems during the installation process.

Reference Platform Definition

The following is defined as the reference platform for the Intel 80x86 (where x is 3 or 4) PC/AT style systems. The software described here is designed for (and has been tested on) this platform.

- i386/i486 processor.
- i80387 floating point unit for i386-based machines.
- At least 8 Mb memory.
- WD1007 compatible disk controller (AT, IDE, ESDI).

- Adaptec 1540/1542 (ISA) or 1742 (EISA) SCSI controller.
- Minimum of 200 Mbytes of disk; 300Mbytes is a better size to store the distribution tree. This should be one of: ESDI, IDE, or SCSI; however, some IDE drives may not work.
- VGA/SVGA controller (e.g. SigmaVGA Legend II or equivalent ET4000) and display.
- WD8003/8013 EP, SMC Elite or Elite16, or Ethercard+ family of Ethernet adapter cards.
- 1.44Mbyte $3^1/_2$" floppy drive.
- While not required, 1.2Mbyte $5^1/_4$" floppy drives are supported.

OSF/1 MK is being used at the OSF Research Institute on the following models of i386/486 AT Clones:

- Toshiba T5200
- HP Vectra series 05/20 RS25C RS20C 486
- Digital DEC PC 450ST
- SNI WX200
- Bull BM600
- Multiplex 486

It is also used on the following multiprocessor machines:

- Digital 433MP (i386 based)
- Compaq SystemPro 486 (i486 based)
- Sequent S81 (i386 based)

Installing The Source

The OSF/1 MK6.1 snapshot consists of one tape containing one *tar*(1) archive. This archive contains the source files and the tools needed to build MK6.1.

Use the following command to unload the source distribution from tape:

`% tar xvf` *norewind-tape-device*

Building The Source

This build process should work on a DEC PC/AT class machine running OSF/1 1.2 or later release (see Reference Platform Definition above). Please note that since the `./src` tree contains all the source files used in the build — it is not necessary to overlay any addtional files onto the MK6.1 tree. Also, neither should you overlay the MK6.1 tree onto an existing OSF 1.X tree since some files in this build are derived from a snapshot of OSF 1.3. A better strategy might be to use the unloaded, complete source tree as a reference and selectively add, delete, or change files to a separate tree as you see fit.

The source tree is extracted using *tar* and created under the `./mk6.1` subdirectory of the current directory:

```
<build_path>/mk6.1
```

where <build_path> is the directory from which you issue the tar command to extract the tape archives.

The extracted mk6.1 tree will contain the following subdirectories:

`./doc` – directory containing release-related documentation

`./export` – tree for headers & libs needed when building

`./obj` – tree for object files, and executable made when building

`./src` – tree containing source code

`./tools` – native tools, scripts, and compiler binaries for building

`./logs` – reference build logs for source (as built by OSF)

The source code provided under `./src` includes only the source files required to build the MK6 kernel and server, the associated software needed for the system to boot, plus Mach-related servers and commands. Note that this source code does not represent a full OSF/1 distribution — in particular, source for commands and libraries is not included.

1. Building The Release

The rest of this section assumes that the contents of the tar archive reside under the directory <build_path>/mk6.1 as described above.

There are a few simple steps to follow when building the source tree. The build process is divided into steps that create MK6 components in the correct order.

If any step fails in the build process below, it is important to identify the error, fix the error, and rebuild, before going on to the next step. Any errors will appear in the log files preceded by the string “`*** Error`”.

Reference logs from each step (as built by OSF) are provided under `<build_path>/mk6.1/logs`. Although pathnames to the build tree and tools will differ, you can generally compare the reference logs to the output from your build logs below.

The build process attempts to use several utilities and libraries installed in the default locations (`/bin`, `/lib`) on your target machine. In particular, the process expects to find the following:

Utilities:

```
cc
lex
yacc
```

Libraries:

```
libc.a
libl.a
libln.a
libm.a
```

The following steps assume you're using a *csh*(1)-like shell; the example target machine to build for is the AT386 platform running OSF/1. (If your target machine is different, then the platform arguments to `build.sh` will differ as well.)

Sources for the FSF `gcc` and the ODE minimal tools set are not included here. Only the binaries for the AT386 platform running OSF/1 are included. If you need to re-build or port these tools for different platforms, please request a tools distribution from the OSF RI.

1. Set your path to pick up the build tools:

```
% path = (<build_path>/mk6.1/tools/at386/hostbin $path)
```

2. Copy the build script from the setup area, and build header and MiG definitions files:

```
% cd <build_path>/mk6.1/src
% cp setup/AT386/build.sh .
% chmod +x build.sh
% cd <build_path>/mk6.1/src
```

This step populates the `./export` subtree with header files. The reference log is `<build_path>/mk6.1/logs/first.log`.

```
% sh -f build.sh AT386 osf1 \
. MAKEFILE_PASS=FIRST >& first.log &
```

```
% tail -f first.log
% grep Error first.log
```

3. Build the microkernel executable file:

The default configuration for the mk6.1 microkernel is FAST. You can change this configuration by setting the environment variable MACH_KERNEL_CONFIG to one of the other available configurations before running build.sh. Common configurations include:

- PRODUCTION – maximal performance, with no debugging capabilities.
- FAST – almost-comparable performance, but with kernel debugger (ddb) and all task symbol tables provided.
- STD+WS – development-oriented, with (e.g.) kernel assertions and lock debugging enabled.
- STD+WS+NORMA – like STD+WS, but with NoRMA-related code enabled.

```
% cd <build_path>/mk6.1/src
% sh -f build.sh AT386 osf1 \
  mach_kernel build_all >& ukernel.log &
% tail -f ukernel.log
% grep Error ukernel.log
```

The reference log is <build_path>/mk6.1/logs/ukernel.log.

4. Build and export the cthreads-aware C runtime startup module:

```
% cd <build_path>/mk6.1/src
% sh -f build.sh AT386 osf1 \
  usr/ccs/lib/crt0 build_all >& crt0.log &
% tail -f crt0.log
% grep Error crt0.log
```

The reference log is <build_path>/mk6.1/logs/crt0.log.

5. Build the libraries used by the server, default_pager and commands:

```
% cd <build_path>/mk6.1/src
% sh -f build.sh AT386 osf1 \
  mach_services build_all >& libs.log &
% tail -f libs.log
% grep Error libs.log
```

After this step, you should find the following files under <build_path>/mk6.1/export/at386/usr/ccs/lib:

```
$ ls <build_path>/mk6.1/export/at386/usr/ccs/lib
```

```
crt0.o          libcthreads_p.a libmachid.a       libsa_mach.a
gcrt0.o         libmach.a        libnetmemory.a   libservice.a
libcthreads.a libmach_p.a       libnetname.a      mcrt0.o
```

This step will also build all available Mach-related servers and commands. The reference log is `<build_path>/mk6.1/logs/libs.log`.

6. Build the bootstrap task:

```
% cd <build_path>/mk6.1/src

% sh -f build.sh AT386 osf1 \
  bootstrap build_all >& bootstrap.log &

% tail -f bootstrap.log

% grep Error bootstrap.log
```

The reference log is `<build_path>/mk6.1/logs/bootstrap.log`.

7. Build the default_pager:

```
% cd <build_path>/mk6.1/src

% sh -f build.sh AT386 osf1 \
  default_pager build_all >& def_pager.log &

% tail -f def_pager.log

% grep Error def_pager.log
```

The reference log is `<build_path>/mk6.1/logs/def_pager.log`.

8. Build the OSF/1 server:

The default configuration for the mk6.1 server is `FAST`. You can change this configuration by setting the environment variable `SERVER_CONFIG` to one of the alternate configurations before running `build.sh`:

- `PRODUCTION` – maximal performance, meant for use in production systems.
- `FAST` – almost-comparable performance, but with development aids provided.
- `DEBUG` – development-oriented, with (e.g.) server assertions and lock debugging enabled.

```
% cd <build_path>/mk6.1/src

% sh -f build.sh AT386 osf1 \
  osf1_server build_all >& server.log &

% tail -f server.log

% grep Error server.log
```

Note the server is built by default to be collocated (loadable into the kernel address space). If you remove the line setting LDFLAGS from the file:

```
<build_path>/mk6.1/src/osf1_server/conf/AT386/template.mk
```

then the server will be built to run in user space. The line you should remove (or comment out) is located after the comment:

```
# make server co-located by default
```

Also note that the FAST and PRODUCTION configurations are currently identical, though this may change in future releases. The reference log is `<build_path>/mk6.1/logs/server.log`.

9. Construct a bootable kernel image:

To be bootable, the kernel executable you built in step (3) must be combined with the bootstrap task built in step (6), in a single pseudo-object file. The utility that does this combining, called `makeboot`, is produced in step (5).

You can find makeboot at:

```
<build_path>/mk6.1/obj/at386/mach_services/cmds/makeboot/make-
boot
```

You can find the bootstrap task at:

```
<build_path>/mk6.1/obj/at386/bootstrap/bootstrap
```

Finally, the kernel executable is produced at:

```
<build_path>/mk6.1/obj/at386/mach_kernel/<kern-config>/mach_-
kernel
```

Given these three files, you construct a bootable kernel image by using makeboot to combine the kernel executable and the bootstrap task with the following command:

```
% makeboot -o mach mach_kernel bootstrap
```

For example, to produce the bootable `mach` image at the top of your object tree, do the following:

```
% cd <build_path>/mk6.1/obj/at386

% mach_services/cmds/makeboot/makeboot -o mach \
  ./mach_kernel/<kern-config>/mach_kernel \
  ./bootstrap/bootstrap
```

This creates <build_path>/mk6.1/obj/at386/mach.

See the next section for directions on how to boot this image on a machine.

2. Booting The Built Images

You must install the microkernel and server images built during the previous steps before booting them. We suggest giving the images names that reflect the configuration they were built with, and the purpose for which the build was done.

So for instance you might copy the bootable kernel produced into the root directory as `/mach.MK61.fast`, and copy the server produced into the `/mach_servers` directory as `osf1_server.MK61.fast`.

Now the new images can be booted by specifying a string like this one at the boot prompt (see the next section):

```
boot: mach.MK61.fast:osf1_server.MK61.fast
```

To set these images to be the defaults, use hard links (or a symbolic link for the server) to install them with the pathnames `/mach` and `/mach_servers/startup`. Naturally, you should save the initial default images under different names before doing this.

```
% ln /mach.MK61.fast /mach

% ln -s /mach_servers/osf1_server.MK61.fast \
  /mach_servers/startup
```

Boot Options

At the boot prompt, rather than booting with the default parameters (booting the microkernel named `/mach`, and starting the server named `/mach_servers/startup`) you may give instructions to the boot program with the following general syntax:

```
disk(disk_unit, disk_partition)/mach_kernel:server -a|-h|-
s|-d|-<digit> server_options
```

where:

a. `disk` is either **hd** for AT, IDE, ESDI disks or **sd** for SCSI disks

b. `disk_unit` is the logical disk unit number from which you want to boot (usually 0 or 1; default is 0).

c. `disk_partition` is the partition containing the root file system (value between `a` to `p`; default is `a`).

d. `mach_kernel` is the name of the file on the root partition containing the microkernel binary you want to boot (default is `mach`). If this name contains no pathname prefix, it will be searched for in the root directory.

e. `server` is the name of the OS server (usually an OSF/1 server) that you want to boot. If this name contains no pathname prefix, it will be searched for in the `/mach_servers` directory.

f. Every field is optional.

and the options shown have these meanings:

`-a` – ask for the server name (to be compatible with old boot)

`-h` – kernel will halt and invoke the debugger as soon as possible (also passed to server)

`-s` – single-user mode (passed to server)

`-d` – load kernel symbols (also done if the `mach_kernel` file has execute permission on)

-`<digit>`– number of CPUs to boot on multiprocessor systems

If you want to boot a non-default server, you must specify the `:server` argument, for example:

```
:osf1_server.mk61.localmods
```

Common Problems

- If you have installed a 80387, make sure that there are no switches to change, or a setup program to be run, to reconfigure the machine.
- Unless you are running X11, OSF/1 MK will use a "monochrome 80 x 25" mode character display. Some VGA cards misbehave; you might get better results with a simple monochrome board if you have one. Also, verify that you have initialized yourVGA board for this mode; some boards require special setup.

Debugging {with} OSF/1 MK

To activate the kernel debugger you must press the `CTRL ALT d` keys simultaneously. In addition to the *ddb* manual pages shipped with this guide, here are some hints:

If an OSF/1 server under debug is being run in user space, then youmay have to tell *ddb* to do "thread-relative" operations when you want to access the server's address space. This tells *ddb* how to access the address space. None of this is relevant to collocated servers (which are the kind produced by default in the build process).

You specify a thread-relative reference by adding the modifier “t” to the *ddb* command you use, and adding the name of the thread to the end of the command line. For instance, the syntax of common examine and write commands becomes:

```
x/<flag>t unix::<varname> $task<n>.<p>o

write/tu unix::<varname> $task<n>.<p> <value>
```

where:

- <flag> is the data-type flag, one of: x, d, i, . . .
- <varname> is the server variable or function name.
- <n> is the task number of the server, as printed by “show all threads”.
- <p> is the number of any thread in the task (0 works).
- <value> is the new value to be assigned to the variable (varname).

Again, the “t” flag, in both commands, indicates that a thread specifier ($task<n>.<p>) appears after the symbol name.

The “u” flag in the write command specifies a reference to user space (i.e., the server is running in user mode and not collocated).

OSF/1 MK Paging

OSF/1 MK uses two paging files:

- The Mach 3.0 microkernel default pager file, as defined in /mach_servers/paging_file (usually a symbolic link).
- The vnode pager file, as defined in /etc/rc.config and/or /etc/fstab.

Any use of anonymous mmap will be backed by the vnode pager (and its paging file). Temporary memory created by Mach services, such as vm_allocate and copy-on-write, will be backed by the default pager (and its paging file).

Since shared-library programs are loaded by use of mmap, they will have their memory backed by the vnode pager. Statically bound programs will be backed by the default pager. Also, there is a special interpretation of the symbolic link found in /mach_servers/paging_file. If it is of the form /dev/hd0b/filename the default pager will use “filename” in the root directory of the file system found on /dev/hd0b. If the symbolic link reads `/dev/hd0b` then the default pager will use the entire partition for raw paging.

When using default paging to a file, you may increase the size of the default pager file by making the file bigger and then rebooting the system. Any additional blocks you add to the paging file will not be used until the next bootstrap.

MK6.1 Known Problems

Introduction

This section describes any known problems remaining in the release at distribution. Problems are listed by affected area: Kernel, Server, Commands, or Unknown (if the particular area has not been determined as yet). The report number for the problem is presented in square brackets at the end of the description (i.e. [ ]).

All known problems are listed here, regardless of which release (microkernel-only or full microkernel/server) is described by this Release Note.

Unknown

- On a two-node cluster running a cnmk_shared STD+WS+NORMA kernel and a fairly old version of the untyped AD1 server, one of the nodes will reliably panic with "thread deallocating itself" before completing two 8-user AIM runs. If one runs the same AD1 server under NMK17.3, the panic does not occur, and multiple AIM runs may be done in succession [594].
- On a two-node cluster, with a STD+WS+NORMA kernel, you can reliably cause a multi-user boot of AD1 to fail by starting the boot process on both clusters almost simultaneously (node 0 a bit before node 1). So long as you get node 1 to complete registration for process creation be-

fore network configuration starts, the boot on node 0 will hang after outputing the hostname of node 0. If you bring up node 0 to multi-user and then add node 1 to the cluster, the failure does not occur [564].

- Presented with an msync() request from VSE, the kernel will hang beneath syscall_vm_region() [555].
- When running two servers kernel-loaded, using the -2 option when starting the second server doesn't work. The second server crashes during the printing of the "Root" message [105].

Kernel

- After a couple of hours of experimenting with the X-server, I found that the load factor on the machine went to 1 and stayed there. There was a zombie thread looping at the end of a short-circuited RPC in the exception function. The kernel debugger claimed that the current task had no threads, so the thread doing this was dead to some major degree. But all appearances to the contrary notwithstanding, it had enough existence left in it to really consume 100% of the CPU (at normal user priority - 12) [622].
- When mach_rpc() functionality is exercised with a MACH_RT configuration kernel, crashes can occur in zalloc() due to a corrupted freelist element pointer [617].
- On an MP system, gcc sometimes dies with 'internal gcc abort'. The process appears to be taking an "invalid operation" FP trap, with status = 82c1, in user mode. This happens when using an MP configuration (which handles fp state less lazily) regardless of the number of cpus started. It does not happen with STD+WS, leading me to suspect the SW FP state handling. Other FP tests have not been tried [616].
- Using the FAST_MP server, which uses finer-grained locking in place of the uni-proc global server lock, causes various failures, presumably in the kernel [615].
- vm_object.h requires inclusion of misc_protos.h to build properly if VM_OBJECT_DEBUG is turned on [612].
- The names for port kotypes in ikot_print_array are not up to date with respect to the types defined in kern/ipc_kobject.h. This leads the kdb command "show port" to print "UNKNOWN" as the type of an activation port, for example [607].
- Any of the mach_perf tests that exercise the async device interfaces will exhaust the zone map in an NMK18 kernel [568].
- The routine syscall_mach_port_allocate_rt fails to check the return code when doing the copyout of the newly-allocated port name. If this

operation fails for some reason, it will still return KERN_SUCCESS [554].

- The kernel as of 8/19/94 cannot boot fully if there are two (as opposed to one) DEC RZ25L disks on the SCSI bus of a DECpc 450st (using an Adaptec 1742 board). It seems that with DEC RZ25L drives, the mode_sense info does not correspond with the actual geometry. It says there are 1872 cylinder's when there are actually closer to 1868 cylinders [469].
- The kernel's copyout (aka copyoutmsg) routine contains a workaround for the i386's assumption that user-read permission implies kernel-write. This is only needed on the i386; the overhead should be avoided on the 486 [301].
- There is currently no prototype definition for thread_switch(). It is not clear exactly where such a prototype should be put [203].
- The pmap_create function should single-thread the memory chunk allocator to avoid multiple allocations. This bug is merely wasteful and then only if NCPUS > 1, hence it is at a low priority [126].
- The fix made for mk6 CR59 to make the short-circuited RPC path honor a MIG_NO_REPLY return from the server will need to be extended once thread_abort() works on migrated threads [120].
- Using the mach_perf performance-measurement program port operations in mk6 appear suspiciously slow [102].
- The mach kernel has several files directly using sys/reboot.h, a unix file. The layering violation should be fixed [97].
- The Adaptec 154x driver is not intelligent about how it copies buffers from the low 16MB of memory to higher memory on EISA-bus machines that can support more than 16MB. Note that this problem will not manifest itself when using Adaptec 174x board's operating in enhanced mode [96].

Server

- The code to forward thread exceptions to the task port (intended for second server debugging with gdb) contains a very embarrassing error. The switch statement to select the exception_raise* variant is missing the break statements that should go after every case, so after forwarding an exception_raise it goes on to try to forward an exception_raise_state, then an exception_raise_state_identity, then panics [618].
- When the X server exits, it closes the mouse and keyboard devices, which causes the kernel to reply to all outstanding read (and write, but that doesn’t matter) requests with D_DEVICE_DOWN. These replies may be received after the device_close call has returned in the server.

Previously this didn’t matter because device_reply_deregister destroyed the device reply port before deallocating the structure that mapped from that port to device-related information. Thus the D_DEVICE_DOWN replies were discarded by the kernel if generated after that point. But now there is no corresponding action, so the server may try to reference a struct dev_reply_entry that has been deallocated [602].

- Server empty activation management is suspect after a heavy load. To try to gather more data (without having to deal with a server containing ~900 activations, its state following a 340-user run), I began running AIM at various levels and trying a "ps ax" afterward. I found that, after AIM at 30 or 60 users, "ps ax" still worked, but that following a 100-user run, it would hang. Analysis shows that one server thread is executing getprocthreads() on behalf of the ps process, and that getprocthreads() is calling thread_info() on one of the server's own threads. This call has hung in mach_msg_receive() [593].
- Booting second servers does not work. The second server gets stuck somewhere after printing the "configure done" message but before reaching single-user mode (this is the point at which multiple server threads start to be created) [591].
- When the server is run in user mode, several zones in the kernel grow without bound, indicating leakage. Ports and various sizes of kalloc'ed things leak rather quickly. To reproduce, boot a user mode server, and snapshot the zone state. Then repeatedly run some command(s) and check the zone state - steady leakage will be seen in several zones. Kernel-loaded servers do not appear to suffer from this. It is suspected that the server is the culprit [538].
- The UNIX exit system call does not work properly when a task contains multiple threads. The result is the task hangs and cannot be killed [484].
- There is a non-deterministic assertion failure in the server. Occasionally the server runs into an assertion failure when booting; apparently a port is being reused [483].
- If you build with lock debugging off (MACH_LDEBUG) and assertions on (MACH_ASSERT) the server does not build [482].
- /dev/zero doesn't exist. The version in AD2 should be adopted [277].
- Server priority depression incompatible with the new version of C-threads. In the new version, blocked C-threads no longer really block, they just depress their priority (with no timeout). Any server thread that depresses itself temporarily allow a different, unblocked server thread to run, will be put in the same priority run queue as all the blocked threads. Even if it is able to make forward progress almost im-

mediately, it will not get any CPU time until all the blocked C-threads in front of it get their time first [135].

- This is OSC CR #10561 and has been fixed OSF/1 R1.3 and should be adopted. It is not high priority but will be fixed when R1.3 is merged into the MK development stream. It describes a potential deadlock in the multithreaded exit handling. Running a test program that create N threads and then exits, will hang the server when N is greater than or equal to 15 [106].
- There are several defects in the malloc/free code fixed via OSC CR #10349 in OSF/1 R1.3 that should be adopted [95].
- The add_device function writes garbage to the d_funnel field of the cdevsw is fixed via OSC CR #9246 from OSF/1 R1.3 and should be adopted [94].
- While running a kernel with all kernel asserts on, the system paniced in the simple unlock code. A code path in the task_dowait routine can try and unlock a simple lock twice in a row. A kernel assert detects this situation and panics the system. This is fixed via OSC CR #10755 from OSF/1 R1.3 and should be adopted [93].
- System took program check 11 because pse_close passed a bad parameter to psa_simple_lock (via csq_acquire). This happened because pse_simple got the parameter private passed to it with all zeros. Further down the trail, it looks like clearalias sets private variable with all zeros in it. This will only occur if clearalias is given a vnode which has a potential alias, but has never been opened (so it has no actual alias). This is fixed via OSC CR #8945 from OSF/1 R1.3 and should be adopted [92].
- The utimes(2) system call contains a chunk of code that is attempting to allow a user that has write access to a file to change the modification time, if it thinks that the user does not own the file. Unfortunately, the interpretation of credentials is the responsibility of the file system code (ufs, nfs, s5fs) via VOP_SETATTR(), and the vfs layer should not be examining uids. This is fixed via OSC CR #9006 from OSF/1 R1.3 and should be adopted [91].
- A race condition between utimes() and sync() occasionally causes the times on a file to be set to the current time instead of the time requested. This is fixed via OSC CR #9985 from OSF/1 R1.3 and should be adopted [90].
- As part of mmap() of a device, the osf1_server (smmap) calls device_pager_create with (dev, mapfun, prot, offset, size); however, the arguments expected by device_pager_create are (dev, offset, size, prot). Remarkably (?) the only visible effect of this seems to be that the MK device_map function gets an inappropriate "prot" value (e.g., 0) which, depending on the checks it makes, may cause it to fail unnecessarily.

This is fixed via OSC CR #10611 from OSF/1 R1.3 and should be adopted [89].

- The kernel functions exit() and fake_u() misuse the NSIG #define in various loops. The index of valid signals goes from 1 to NSIG, but the logic in these functions loops, incorrectly, from signal 0 to NSIG-1. This is fixed via OSC CR #10409 from OSF/1 R1.3 and should be adopted [88].
- The task_get_rusage() routine sends the task_info() Mach message twice (once in pmap_max_resident_count() and once in pmap_integrated_rss()). In neither case does the kernel provide the information needed to compute the required statistics. This is fixed via OSC CR #10908 from OSF/1 R1.3 and should be adopted [87].
- A number of improvements have been made in OSF/1 R1.3 to tty handling in both server and kernel, which should be adopted. Notable among these are the addition of modem control support and the fixing of a race condition that occasionally caused a memory fault during console opens when booting [86].
- In scandir brelse() while checking the validity of a directory entry after it has been found as a match, a reference is made to the size of the directory entry after the structure (buf) which contains this information has been freed. This is fixed via OSC CR #8668 from OSF/1 R1.3 and should be adopted [84].
- AST prevents thread state changes; during the pthread server syscall port the async cancellation of a blocking system call was attempted and failed. This is fixed via OSC CR #9719 from OSF/1 R1.3 and should be adopted [83].
- The definition of microtime() is subtly different between the IK & MK. The "unique timestamps" code has been shown to cause problems in the past, and should be removed from the MK. This is fixed via OSC CR #4317 from OSF/1 R1.3 and should be adopted [82].
- The mk6 code base needs to pick up the 1.3 modem control changes. There are several problems, along with RTS/CTS support, which is an enhancement. This is fixed via OSC CR #9843 from OSF/1 R1.3 and should be adopted [71].

Commands

- I currently have two gdb binaries for debugging Mach programs. They are gdb-rose-untyped and gdb-coloc. The latter has been adapted to understand the new cthreads library so in principle it should work better with our servers. But in practice any significant use of it causes it to crash out with a memory allocation failure [614].

- Using the -d option to pinfo (on an ordinary user-mode process) does-not work. This is the option that uses mach_port_space_info() to query IPC space usage data [103].
- The mach_perf command should be ported to mk6. Doing so raises at least one issue -- port renaming [99].

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

```
struct mach_port_status
{
        mach_port_name_t          mps_pset;
        mach_port_seqno_t         mps_seqno;
        mach_port_mscount_t       mps_mscount;
        mach_port_msgcount_t      mps_msgcount;
        mach_port_rights_t        mps_sorights;
        boolean_t                 mps_srights;
        boolean_t                 mps_nsrequest;
        boolean_t                 mps_rt;
};
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

```
struct routine_descriptor
{
        mig_impl_routine_t              impl_routine;
        mig_stub_routine_t              stub_routine;
        unsigned int                    argc;
        unsigned int                    descr_count;
        struct routine_arg_descriptor   *arg_descr;
        unsigned int                    max_reply_msg;
} *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

```
struct rpc_subsystem
{
        struct subsystem                  *subsystem;
        mach_msg_id_t                     start;
        mach_msg_id_t                     end;
        unsigned int                      maxsize;
        vm_address_t                      base_addr;
        struct routine_descriptor         routine[1];
        struct routine_arg_descriptor     arg_descriptor[1];
} *rpc_subsystem_t;


#define RPC_SUBSYSTEM_NULL                ((rpc_subsystem_t) 0)
```

DESCRIPTION

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

<u>Note:</u> 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

```
struct io_done_result
{
struct control_info
{
        int                         qd_type;
        mach_port_t                 qd_reqid;
        kern_return_t               qd_code;
        io_buf_ptr_t                qd_data;
        int                         qd_count;
} qd_control;
io_buf_ptr_inband_t                 qd_inline;
};
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**.