The GNU Mach Reference Manual

The GNU Mach Reference Manual

Marcus Brinkmann
with
Gordon Matzigkeit, Gibran Hasnaoui,
Robert V. Baron, Richard P. Draves, Mary R. Thompson, Joseph S. Barrera

Edition 0.4

last updated 2001-09-01

for version 1.2

1 Introduction

GNU Mach is the microkernel of the GNU Project. It is the base of the operating system, and provides its functionality to the Hurd servers, the GNU C Library and all user applications. The microkernel itself does not provide much functionality of the system, just enough to make it possible for the Hurd servers and the C library to implement the missing features you would expect from a POSIX compatible operating system.

1.1 Audience

This manual is designed to be useful to everybody who is interested in using, administering, or programming the Mach microkernel.

If you are an end-user and you are looking for help on running the Mach kernel, the first few chapters of this manual describe the essential parts of installing and using the kernel in the GNU operating system.

The rest of this manual is a technical discussion of the Mach programming interface and its implementation, and would not be helpful until you want to learn how to extend the system or modify the kernel.

This manual is organized according to the subsystems of Mach, and each chapter begins with descriptions of conceptual ideas that are related to that subsystem. If you are a programmer and want to learn more about, say, the Mach IPC subsystem, you can skip to the IPC chapter (see Chapter 4 [Inter Process Communication], page 13), and read about the related concepts and interface definitions.

1.2 Features

GNU Mach is not the most advanced microkernel known to the planet, nor is it the fastest or smallest, but it has a rich set of interfaces and some features which make it useful as the base of the Hurd system.

it's free software
: Anybody can use, modify, and redistribute it under the terms of the GNU General Public License (see Appendix A [Copying], page 113). GNU Mach is part of the GNU system, which is a complete operating system licensed under the GPL.

it's built to survive
: As a microkernel, GNU Mach doesn't implement a lot of the features commonly found in an operating system, but only the bare minimum that is required to implement a full operating system on top of it. This means that a lot of the operating system code is maintained outside of GNU Mach, and while this code may go through a complete redesign, the code of the microkernel can remain comparatively stable.

it's scalable
: Mach is particularly well suited for SMP and network cluster techniques. Thread support is provided at the kernel level, and the kernel itself takes advantage of that. Network transparency at the IPC level makes resources of

the system available across machine boundaries (with NORMA IPC, currently not available in GNU Mach).

it exists
: The Mach microkernel is real software that works Right Now. It is not a research or a proposal. You don't have to wait at all before you can start using and developing it. Mach has been used in many operating systems in the past, usually as the base for a single UNIX server. In the GNU system, Mach is the base of a functional multi-server operating system, the Hurd.

1.3 Overview

An operating system kernel provides a framework for programs to share a computer's hardware resources securely and efficiently. This requires that the programs are seperated and protected from each other. To make running multiple programs in parallel useful, there also needs to be a facility for programs to exchange information by communication.

The Mach microkernel provides abstractions of the underlying hardware resources like devices and memory. It organizes the running programs into tasks and threads (points of execution in the tasks). In addition, Mach provides a rich interface for inter-process communication.

What Mach does not provide is a POSIX compatible programming interface. In fact, it has no understanding of file systems, POSIX process semantics, network protocols and many more. All this is implemented in tasks running on top of the microkernel. In the GNU operating system, the Hurd servers and the C library share the responsibility to implement the POSIX interface, and the additional interfaces which are specific to the GNU system.

1.4 History

XXX A few lines about the history of Mach here.

2 Installing

Before you can use the Mach microkernel in your system you'll need to install it and all components you want to use with it, e.g. the rest of the operating system. You also need a bootloader to load the kernel from the storage medium and run it when the computer is started.

GNU Mach is only available for Intel i386-compatible architectures (such as the Pentium) currently. If you have a different architecture and want to run the GNU Mach microkernel, you will need to port the kernel and all other software of the system to your machine's architecture. Porting is an involved process which requires considerable programming skills, and it is not recommended for the faint-of-heart. If you have the talent and desire to do a port, contact `bug-hurd@gnu.org` in order to coordinate the effort.

2.1 Binary Distributions

By far the easiest and best way to install GNU Mach and the operating system is to obtain a GNU binary distribution. The GNU operating system consists of GNU Mach, the Hurd, the C library and many applications. Without the GNU operating system, you will only have a microkernel, which is not very useful by itself, without the other programs.

Building the whole operating system takes a huge effort, and you are well advised to not do it yourself, but to get a binary distribution of the GNU operating system. The distribution also includes a binary of the GNU Mach microkernel.

Information on how to obtain the GNU system can be found in the Hurd info manual.

2.2 Compilation

If you already have a running GNU system, and only want to recompile the kernel, for example to select a different set of included hardware drivers, you can easily do this. You need the GNU C compiler and MiG, the Mach interface generator, which both come in their own packages.

Building and installing the kernel is as easy as with any other GNU software package. The configure script is used to configure the source and set the compile time options. The compilation is done by running:

```
make
```

To install the kernel and its header files, just enter the command:

```
make install
```

This will install the kernel into $(prefix)/boot/gnumach and the header files into $(prefix)/include. You can also only install the kernel or the header files. For this, the two targets install-kernel and install-headers are provided.

2.3 Configuration

The following options can be passed to the configure script as command line arguments and control what components are built into the kernel, or where it is installed.

The default for an option is to be disabled, unless otherwise noted.

`--prefix` *prefix*
: Sets the prefix to PREFIX. The default prefix is the empty string, which is the correct value for the GNU system. The prefix is prepended to all file names at installation time.

`--enable-kdb`
: Enables the in-kernel debugger. This is only useful if you actually anticipate debugging the kernel. It is not enabled by default because it adds considerably to the unpageable memory footprint of the kernel. See Chapter 11 [Kernel Debugger], page 105.

`--enable-kmsg`
: Enables the kernel message device kmsg.

`--enable-lpr`
: Enables the parallel port devices lpr%d.

`--enable-floppy`
: Enables the PC floppy disk controller devices fd%d.

`--enable-ide`
: Enables the IDE controller devices hd%d, hd%ds%d.

The following options enable drivers for various SCSI controller. SCSI devices are named sd%d (disks) or cd%d (CD ROMs).

`--enable-advansys`
: Enables the AdvanSys SCSI controller devices sd%d, cd%d.

`--enable-buslogic`
: Enables the BusLogic SCSI controller devices sd%d, cd%d.

`--disable-flashpoint`
: Only meaningful in conjunction with '`--enable-buslogic`'. Omits the Flsh-Point support. This option is enabled by default if '`--enable-buslogic`' is specified.

`--enable-u1434f`
: Enables the UltraStor 14F/34F SCSI controller devices sd%d, cd%d.

`--enable-ultrastor`
: Enables the UltraStor SCSI controller devices sd%d, cd%d.

`--enable-aha152x`
`--enable-aha2825`
: Enables the Adaptec AHA-152x/2825 SCSI controller devices sd%d, cd%d.

`--enable-aha1542`
: Enables the Adaptec AHA-1542 SCSI controller devices sd%d, cd%d.

`--enable-aha1740`
: Enables the Adaptec AHA-1740 SCSI controller devices sd%d, cd%d.

`--enable-aic7xxx`
: Enables the Adaptec AIC7xxx SCSI controller devices sd%d, cd%d.

`--enable-futuredomain`
: Enables the Future Domain 16xx SCSI controller devices sd%d, cd%d.

`--enable-in2000`
: Enables the Always IN 2000 SCSI controller devices sd%d, cd%d.

`--enable-ncr5380`
`--enable-ncr53c400`
: Enables the generic NCR5380/53c400 SCSI controller devices sd%d, cd%d.

`--enable-ncr53c406a`
: Enables the NCR53c406a SCSI controller devices sd%d, cd%d.

`--enable-pas16`
: Enables the PAS16 SCSI controller devices sd%d, cd%d.

`--enable-seagate`
: Enables the Seagate ST02 and Future Domain TMC-8xx SCSI controller devices sd%d, cd%d.

`--enable-t128`
`--enable-t128f`
`--enable-t228`
: Enables the Trantor T128/T128F/T228 SCSI controller devices sd%d, cd%d.

`--enable-ncr53c7xx`
: Enables the NCR53C7,8xx SCSI controller devices sd%d, cd%d.

`--enable-eatadma`
: Enables the EATA-DMA (DPT, NEC, AT&T, SNI, AST, Olivetti, Alphatronix) SCSI controller devices sd%d, cd%d.

`--enable-eatapio`
: Enables the EATA-PIO (old DPT PM2001, PM2012A) SCSI controller devices sd%d, cd%d.

`--enable-wd7000`
: Enables the WD 7000 SCSI controller devices sd%d, cd%d.

`--enable-eata`
: Enables the EATA ISA/EISA/PCI (DPT and generic EATA/DMA-compliant boards) SCSI controller devices sd%d, cd%d.

`--enable-am53c974`
`--enable-am79c974`
: Enables the AM53/79C974 SCSI controller devices sd%d, cd%d.

`--enable-dtc3280`
`--enable-dtc3180`
: Enables the DTC3180/3280 SCSI controller devices sd%d, cd%d.

`--enable-ncr53c8xx`
`--enable-dc390w`
`--enable-dc390u`
`--enable-dc390f`
: Enables the NCR53C8XX SCSI controller devices sd%d, cd%d.

`--enable-dc390t`
`--enable-dc390`
: Enables the Tekram DC-390(T) SCSI controller devices sd%d, cd%d.

`--enable-ppa`
: Enables the IOMEGA Parallel Port ZIP drive device sd%d.

`--enable-qlogicfas`
: Enables the Qlogic FAS SCSI controller devices sd%d, cd%d.

`--enable-qlogicisp`
: Enables the Qlogic ISP SCSI controller devices sd%d, cd%d.

`--enable-gdth`
: Enables the GDT SCSI Disk Array controller devices sd%d, cd%d.

The following options enable drivers for various ethernet cards. NIC device names are usually eth%d, except for the pocket adaptors.

GNU Mach does only autodetect one ethernet card. To enable any further cards, the source code has to be edited.

`--enable-ne2000`
`--enable-ne1000`
: Enables the NE2000/NE1000 ISA netword card devices eth%d.

`--enable-3c503`
`--enable-el2`
: Enables the 3Com 503 (Etherlink II) netword card devices eth%d.

`--enable-3c509`
`--enable-3c579`
`--enable-el3`
: Enables the 3Com 509/579 (Etherlink III) netword card devices eth%d.

`--enable-wd80x3`
: Enables the WD80X3 netword card devices eth%d.

`--enable-3c501`
`--enable-el1`
: Enables the 3COM 501 netword card devices eth%d.

`--enable-ul`
: Enables the SMC Ultra netword card devices eth%d.

`--enable-ul32`
: Enables the SMC Ultra 32 netword card devices eth%d.

`--enable-hplanplus`
: Enables the HP PCLAN+ (27247B and 27252A) netword card devices eth%d.

`--enable-hplan`
: Enables the HP PCLAN (27245 and other 27xxx series) netword card devices eth%d.

`--enable-3c59x`
`--enable-3c90x`
`--enable-vortex`
: Enables the 3Com 590/900 series (592/595/597/900/905) "Vortex/Boomerang" netword card devices eth%d.

`--enable-seeq8005`
: Enables the Seeq8005 netword card devices eth%d.

`--enable-hp100`
`--enable-hpj2577`
`--enable-hpj2573`
`--enable-hp27248b`
`--enable-hp2585`
: Enables the HP 10/100VG PCLAN (ISA, EISA, PCI) netword card devices eth%d.

`--enable-ac3200`
: Enables the Ansel Communications EISA 3200 netword card devices eth%d.

`--enable-e2100`
: Enables the Cabletron E21xx netword card devices eth%d.

`--enable-at1700`
: Enables the AT1700 (Fujitsu 86965) netword card devices eth%d.

`--enable-eth16i`
`--enable-eth32`
: Enables the ICL EtherTeam 16i/32 netword card devices eth%d.

`--enable-znet`
`--enable-znote`
: Enables the Zenith Z-Note netword card devices eth%d.

`--enable-eexpress`
: Enables the EtherExpress 16 netword card devices eth%d.

`--enable-eexpresspro`
: Enables the EtherExpressPro netword card devices eth%d.

`--enable-eexpresspro100`
: Enables the Intel EtherExpressPro PCI 10+/100B/100+ netword card devices eth%d.

`--enable-depca`
`--enable-de100`
`--enable-de101`
`--enable-de200`
`--enable-de201`
`--enable-de202`
`--enable-de210`
`--enable-de422`
: Enables the DEPCA, DE10x, DE200, DE201, DE202, DE210, DE422 netword card devices eth%d.

`--enable-ewrk3`
`--enable-de203`
`--enable-de204`
`--enable-de205`
: Enables the EtherWORKS 3 (DE203, DE204, DE205) netword card devices eth%d.

`--enable-de4x5`
`--enable-de425`
`--enable-de434`
`--enable-435`
`--enable-de450`
`--enable-500`
: Enables the DE425, DE434, DE435, DE450, DE500 netword card devices eth%d.

`--enable-apricot`
: Enables the Apricot XEN-II on board ethernet netword card devices eth%d.

`--enable-wavelan`
: Enables the AT&T WaveLAN & DEC RoamAbout DS netword card devices eth%d.

`--enable-3c507`
`--enable-el16`
: Enables the 3Com 507 netword card devices eth%d.

`--enable-3c505`
`--enable-elplus`
: Enables the 3Com 505 netword card devices eth%d.

`--enable-de600`
: Enables the D-Link DE-600 netword card devices eth%d.

`--enable-de620`
: Enables the D-Link DE-620 netword card devices eth%d.

`--enable-skg16`
: Enables the Schneider & Koch G16 netword card devices eth%d.

`--enable-ni52`
: Enables the NI5210 netword card devices eth%d.

`--enable-ni65`
: Enables the NI6510 netword card devices eth%d.

`--enable-atp`
: Enables the AT-LAN-TEC/RealTek pocket adaptor netword card devices atp%d.

`--enable-lance`
`--enable-at1500`
`--enable-ne2100`
: Enables the AMD LANCE and PCnet (AT1500 and NE2100) netword card devices eth%d.

`--enable-elcp`
`--enable-tulip`
Enables the DECchip Tulip (dc21x4x) PCI netword card devices eth%d.

`--enable-fmv18x`
Enables the FMV-181/182/183/184 netword card devices eth%d.

`--enable-3c515`
Enables the 3Com 515 ISA Fast EtherLink netword card devices eth%d.

`--enable-pcnet32`
Enables the AMD PCI PCnet32 (PCI bus NE2100 cards) netword card devices eth%d.

`--enable-ne2kpci`
Enables the PCI NE2000 netword card devices eth%d.

`--enable-yellowfin`
Enables the Packet Engines Yellowfin Gigabit-NIC netword card devices eth%d.

`--enable-rtl8139`
`--enable-rtl8129`
Enables the RealTek 8129/8139 (not 8019/8029!) netword card devices eth%d.

`--enable-epic`
`--enable-epic100`
Enables the SMC 83c170/175 EPIC/100 (EtherPower II) netword card devices eth%d.

`--enable-tlan`
Enables the TI ThunderLAN netword card devices eth%d.

`--enable-viarhine`
Enables the VIA Rhine netword card devices eth%d.

2.4 Cross-Compilation

Another way to install the kernel is to use an existing operating system in order to compile the kernel binary. This is called *cross-compiling*, because it is done between two different platforms. If the pre-built kernels are not working for you, and you can't ask someone to compile a custom kernel for your machine, this is your last chance to get a kernel that boots on your hardware.

Luckily, the kernel does have light dependencies. You don't even need a cross compiler if your build machine has a compiler and is the same architecture as the system you want to run GNU Mach on.

You need a cross-mig, though.

XXX More info needed.

3 Bootstrap

Bootstrapping[1] is the procedure by which your machine loads the microkernel and transfers control to the operating system.

3.1 Bootloader

The *bootloader* is the first software that runs on your machine. Many hardware architectures have a very simple startup routine which reads a very simple bootloader from the beginning of the internal hard disk, then transfers control to it. Other architectures have startup routines which are able to understand more of the contents of the hard disk, and directly start a more advanced bootloader.

Currently, *GRUB*[2] is the preferred GNU bootloader. GRUB provides advanced functionality, and is capable of loading several different kernels (such as Mach, Linux, DOS, and the *BSD family). See section "Introduction" in *GRUB Manual*.

GNU Mach conforms to the Multiboot specification which defines an interface between the bootloader and the components that run very early at startup. GNU Mach can be started by any bootloader which supports the multiboot standard. After the bootloader loaded the kernel image to a designated address in the system memory, it jumps into the startup code of the kernel. This code initializes the kernel and detects the available hardware devices. Afterwards, the first system task is started. See section "Overview" in *Multiboot Specification*.

3.2 Modules

Because the microkernel does not provide filesystem support and other features necessary to load the first system task from a storage medium, the first task is loaded by the bootloader as a module to a specified address. In the GNU system, this first program is the `serverboot` executable. GNU Mach inserts the host control port and the device master port into this task and appends the port numbers to the command line before executing it.

The `serverboot` program is responsible for loading and executing the rest of the Hurd servers. Rather than containing specific instructions for starting the Hurd, it follows general steps given in a user-supplied boot script.

XXX More about boot scripts.

---

[1] The term *bootstrapping* refers to a Dutch legend about a boy who was able to fly by pulling himself up by his bootstraps. In computers, this term refers to any process where a simple system activates a more complicated system.

[2] The GRand Unified Bootloader, available from `http://www.uruk.org/grub/`.

4 Inter Process Communication

This chapter describes the details of the Mach IPC system. First the actual calls concerned with sending and receiving messages are discussed, then the details of the port system are described in detail.

4.1 Major Concepts

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

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

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

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

Ports hold a queue of messages. Tasks operate on a port to send and receive messages by exercising capabilities for the port. Multiple tasks can hold send capabilities, or rights, for a port. Tasks can also hold send-once rights, which grant the ability to send a single message. Only one task can hold the receive capability, or receive right, for a port. Port rights can be transferred between tasks via messages. The sender of a message can specify in the message body that the message contains a port right. If a message contains a receive right for a port, then the receive right is removed from the sender of the message and the right is transferred to the receiver of the message. While the receive right is in transit, tasks holding send rights can still send messages to the port, and they are queued until a task acquires the receive right and uses it to receive the messages.

Tasks can receive messages from ports and port sets. The port set abstraction allows a single thread to wait for a message from any of several ports. Tasks manipulate port sets with a capability, or port-set right, which is taken from the same space as the port capabilities. The port-set right may not be transferred in a message. A port set holds receive rights, and a receive operation on a port set blocks waiting for a message sent to any of the constituent ports. A port may not belong to more than one port set, and if a port is a member of a port set, the holder of the receive right can't receive directly from the port.

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

4.2 Messaging Interface

This section describes how messages are composed, sent and received within the Mach IPC system.

4.2.1 Mach Message Call

To use the `mach_msg` call, you can include the header files '`mach/port.h`' and '`mach/message.h`'.

`mach_msg_return_t mach_msg` (*mach_msg_header_t* `*msg`, *mach_msg_option_t* `option`, *mach_msg_size_t* `send_size`, *mach_msg_size_t* `rcv_size`, *mach_port_t* `rcv_name`, *mach_msg_timeout_t* `timeout`, *mach_port_t* `notify`) [Function]

The `mach_msg` function is used to send and receive messages. Mach messages contain typed data, which can include port rights and references to large regions of memory.

msg is the address of a buffer in the caller's address space. Message buffers should be aligned on long-word boundaries. The message options *option* are bit values, combined with bitwise-or. One or both of `MACH_SEND_MSG` and `MACH_RCV_MSG` should be used. Other options act as modifiers. When sending a message, *send_size* specifies the size of the message buffer. Otherwise zero should be supplied. When receiving a message, *rcv_size* specifies the size of the message buffer. Otherwise zero should be supplied. When receiving a message, *rcv_name* specifies the port or port set. Otherwise `MACH_PORT_NULL` should be supplied. When using the `MACH_SEND_TIMEOUT` and `MACH_RCV_TIMEOUT` options, *timeout* specifies the time in milliseconds to wait before giving up. Otherwise `MACH_MSG_TIMEOUT_NONE` should be supplied. When using the `MACH_SEND_NOTIFY`, `MACH_SEND_CANCEL`, and `MACH_RCV_NOTIFY` options, *notify* specifies the port used for the notification. Otherwise `MACH_PORT_NULL` should be supplied.

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

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

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

If the option argument specifies neither `MACH_SEND_MSG` nor `MACH_RCV_MSG`, then `mach_msg` does nothing.

Some options, like `MACH_SEND_TIMEOUT` and `MACH_RCV_TIMEOUT`, share a supporting argument. If these options are used together, they make independent use of the supporting argument's value.

`mach_msg_timeout_t` [Data type]

This is a `natural_t` used by the timeout mechanism. The units are milliseconds. The value to be used when there is no timeout is `MACH_MSG_TIMEOUT_NONE`.

4.2.2 Message Format

A Mach message consists of a fixed size message header, a `mach_msg_header_t`, followed by zero or more data items. Data items are typed. Each item has a type descriptor followed by the actual data (or the address of the data, for out-of-line memory regions).

The following data types are related to Mach ports:

`mach_port_t` [Data type]

The `mach_port_t` data type is an unsigned integer type which represents a port name in the task's port name space. In GNU Mach, this is an `unsigned int`.

The following data types are related to Mach messages:

`mach_msg_bits_t` [Data type]

The `mach_msg_bits_t` data type is an `unsigned int` used to store various flags for a message.

`mach_msg_size_t` [Data type]

The `mach_msg_size_t` data type is an `unsigned int` used to store the size of a message.

`mach_msg_id_t` [Data type]

The `mach_msg_id_t` data type is an `integer_t` typically used to convey a function or operation id for the receiver.

`mach_msg_header_t` [Data type]

This structure is the start of every message in the Mach IPC system. It has the following members:

`mach_msg_bits_t msgh_bits`

The `msgh_bits` field has the following bits defined, all other bits should be zero:

`MACH_MSGH_BITS_REMOTE_MASK`
`MACH_MSGH_BITS_LOCAL_MASK`

The remote and local bits encode `mach_msg_type_name_t` values that specify the port rights in the `msgh_remote_port` and `msgh_local_port` fields. The remote value must specify a send or send-once right for the destination of the message. If the local value doesn't specify a send or send-once right for the message's reply port, it must be zero and msgh_local_port must be `MACH_PORT_NULL`.

`MACH_MSGH_BITS_COMPLEX`
: The complex bit must be specified if the message body contains port rights or out-of-line memory regions. If it is not specified, then the message body carries no port rights or memory, no matter what the type descriptors may seem to indicate.

`MACH_MSGH_BITS_REMOTE` and `MACH_MSGH_BITS_LOCAL` macros return the appropriate `mach_msg_type_name_t` values, given a `msgh_bits` value. The `MACH_MSGH_BITS` macro constructs a value for `msgh_bits`, given two `mach_msg_type_name_t` values.

`mach_msg_size_t msgh_size`
: The `msgh_size` field in the header of a received message contains the message's size. The message size, a byte quantity, includes the message header, type descriptors, and in-line data. For out-of-line memory regions, the message size includes the size of the in-line address, not the size of the actual memory region. There are no arbitrary limits on the size of a Mach message, the number of data items in a message, or the size of the data items.

`mach_port_t msgh_remote_port`
: The `msgh_remote_port` field specifies the destination port of the message. The field must carry a legitimate send or send-once right for a port.

`mach_port_t msgh_local_port`
: The `msgh_local_port` field specifies an auxiliary port right, which is conventionally used as a reply port by the recipient of the message. The field must carry a send right, a send-once right, `MACH_PORT_NULL`, or `MACH_PORT_DEAD`.

`mach_port_seqno_t msgh_seqno`
: The `msgh_seqno` field provides a sequence number for the message. It is only valid in received messages; its value in sent messages is overwritten.

`mach_msg_id_t msgh_id`
: The `mach_msg` call doesn't use the `msgh_id` field, but it conventionally conveys an operation or function id.

`mach_msg_bits_t MACH_MSGH_BITS` (*mach_msg_type_name_t* `remote`, *mach_msg_type_name_t* `local`) [Macro]

This macro composes two `mach_msg_type_name_t` values that specify the port rights in the `msgh_remote_port` and `msgh_local_port` fields of a `mach_msg` call into an appropriate `mach_msg_bits_t` value.

`mach_msg_type_name_t MACH_MSGH_BITS_REMOTE (`*mach_msg_bits_t* `bits)` [Macro]

This macro extracts the `mach_msg_type_name_t` value for the remote port right in a `mach_msg_bits_t` value.

`mach_msg_type_name_t MACH_MSGH_BITS_LOCAL (`*mach_msg_bits_t* `bits)` [Macro]

This macro extracts the `mach_msg_type_name_t` value for the local port right in a `mach_msg_bits_t` value.

`mach_msg_bits_t MACH_MSGH_BITS_PORTS (`*mach_msg_bits_t* `bits)` [Macro]

This macro extracts the `mach_msg_bits_t` component consisting of the `mach_msg_type_name_t` values for the remote and local port right in a `mach_msg_bits_t` value.

`mach_msg_bits_t MACH_MSGH_BITS_OTHER (`*mach_msg_bits_t* `bits)` [Macro]

This macro extracts the `mach_msg_bits_t` component consisting of everything except the `mach_msg_type_name_t` values for the remote and local port right in a `mach_msg_bits_t` value.

Each data item has a type descriptor, a `mach_msg_type_t` or a `mach_msg_type_long_t`. The `mach_msg_type_long_t` type descriptor allows larger values for some fields. The `msgtl_header` field in the long descriptor is only used for its inline, longform, and deallocate bits.

`mach_msg_type_name_t` [Data type]

This is an `unsigned int` and can be used to hold the `msgt_name` component of the `mach_msg_type_t` and `mach_msg_type_long_t` structure.

`mach_msg_type_size_t` [Data type]

This is an `unsigned int` and can be used to hold the `msgt_size` component of the `mach_msg_type_t` and `mach_msg_type_long_t` structure.

`mach_msg_type_number_t` [Data type]

This is an `natural_t` and can be used to hold the `msgt_number` component of the `mach_msg_type_t` and `mach_msg_type_long_t` structure.

`mach_msg_type_t` [Data type]

This structure has the following members:

`unsigned int msgt_name : 8`

The `msgt_name` field specifies the data's type. The following types are predefined:

`MACH_MSG_TYPE_UNSTRUCTURED`
`MACH_MSG_TYPE_BIT`
`MACH_MSG_TYPE_BOOLEAN`
`MACH_MSG_TYPE_INTEGER_16`
`MACH_MSG_TYPE_INTEGER_32`
`MACH_MSG_TYPE_CHAR`
`MACH_MSG_TYPE_BYTE`
`MACH_MSG_TYPE_INTEGER_8`
`MACH_MSG_TYPE_REAL`
`MACH_MSG_TYPE_STRING`
`MACH_MSG_TYPE_STRING_C`
`MACH_MSG_TYPE_PORT_NAME`

The following predefined types specify port rights, and receive special treatment. The next section discusses these types in detail. The type `MACH_MSG_TYPE_PORT_NAME` describes port right names, when no rights are being transferred, but just names. For this purpose, it should be used in preference to `MACH_MSG_TYPE_INTEGER_32`.

`MACH_MSG_TYPE_MOVE_RECEIVE`
`MACH_MSG_TYPE_MOVE_SEND`
`MACH_MSG_TYPE_MOVE_SEND_ONCE`
`MACH_MSG_TYPE_COPY_SEND`
`MACH_MSG_TYPE_MAKE_SEND`
`MACH_MSG_TYPE_MAKE_SEND_ONCE`

`msgt_size : 8`
: The `msgt_size` field specifies the size of each datum, in bits. For example, the msgt_size of `MACH_MSG_TYPE_INTEGER_32` data is 32.

`msgt_number : 12`
: The `msgt_number` field specifies how many data elements comprise the data item. Zero is a legitimate number.

: The total length specified by a type descriptor is `(msgt_size * msgt_number)`, rounded up to an integral number of bytes. In-line data is then padded to an integral number of long-words. This ensures that type descriptors always start on long-word boundaries. It implies that message sizes are always an integral multiple of a long-word's size.

`msgt_inline : 1`
: The `msgt_inline` bit specifies, when `FALSE`, that the data actually resides in an out-of-line region. The address of the memory region (a `vm_offset_t` or `vm_address_t`) follows the type descriptor in the message body. The `msgt_name`, `msgt_size`, and `msgt_number` fields describe the memory region, not the address.

`msgt_longform : 1`
: The `msgt_longform` bit specifies, when `TRUE`, that this type descriptor is a `mach_msg_type_long_t` instead of a `mach_msg_type_t`. The `msgt_name`, `msgt_size`, and `msgt_number` fields should be zero. Instead, `mach_msg` uses the following `msgtl_name`, `msgtl_size`, and `msgtl_number` fields.

`msgt_deallocate : 1`
The `msgt_deallocate` bit is used with out-of-line regions. When `TRUE`, it specifies that the memory region should be deallocated from the sender's address space (as if with `vm_deallocate`) when the message is sent.

`msgt_unused : 1`
The `msgt_unused` bit should be zero.

`boolean_t MACH_MSG_TYPE_PORT_ANY` (*mach_msg_type_name_t type*) [Macro]
This macro returns `TRUE` if the given type name specifies a port type, otherwise it returns `FALSE`.

`boolean_t MACH_MSG_TYPE_PORT_ANY_SEND` (*mach_msg_type_name_t type*) [Macro]
This macro returns `TRUE` if the given type name specifies a port type with a send or send-once right, otherwise it returns `FALSE`.

`boolean_t MACH_MSG_TYPE_PORT_ANY_RIGHT` (*mach_msg_type_name_t type*) [Macro]
This macro returns `TRUE` if the given type name specifies a port right type which is moved, otherwise it returns `FALSE`.

`mach_msg_type_long_t` [Data type]
This structure has the following members:

`mach_msg_type_t msgtl_header`
Same meaning as `msgt_header`.

`unsigned short msgtl_name`
Same meaning as `msgt_name`.

`unsigned short msgtl_size`
Same meaning as `msgt_size`.

`unsigned int msgtl_number`
Same meaning as `msgt_number`.

4.2.3 Exchanging Port Rights

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

There are five basic kinds of rights: receive rights, send rights, send-once rights, port-set rights, and dead names. Dead names are not capabilities. They act as place-holders to prevent a name from being otherwise used.

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

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

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

A message body can carry port rights; the `msgt_name` (`msgtl_name`) field in a type descriptor specifies the type of port right and how the port right is to be extracted from the caller. The values `MACH_PORT_NULL` and `MACH_PORT_DEAD` are always valid in place of a port right in a message body. In a sent message, the following `msgt_name` values denote port rights:

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

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

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

`MACH_MSG_TYPE_MAKE_SEND_ONCE`
: The message will carry a send-once right, but the caller must supply a receive right. The send-once right is created from the receive right.

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

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

If a message carries a send or send-once right, and the port dies while the message is in transit, then the receiving task will get `MACH_PORT_DEAD` instead of a right. The following `msgt_name` values in a received message indicate that it carries port rights:

`MACH_MSG_TYPE_PORT_SEND`
: This name is an alias for `MACH_MSG_TYPE_MOVE_SEND`. The message carried a send right. If the receiving task already has send and/or receive rights for the

port, then that name for the port will be reused. Otherwise, the new right will have a new name. If the task already has send rights, it gains a user reference for the right (unless this would cause the user-reference count to overflow). Otherwise, it acquires the send right, with a user-reference count of one.

`MACH_MSG_TYPE_PORT_SEND_ONCE`
: This name is an alias for `MACH_MSG_TYPE_MOVE_SEND_ONCE`. The message carried a send-once right. The right will have a new name.

`MACH_MSG_TYPE_PORT_RECEIVE`
: This name is an alias for `MACH_MSG_TYPE_MOVE_RECEIVE`. The message carried a receive right. If the receiving task already has send rights for the port, then that name for the port will be reused. Otherwise, the right will have a new name. The make-send count of the receive right is reset to zero, but the port retains other attributes like queued messages, extant send and send-once rights, and requests for port-destroyed and no-senders notifications.

When the kernel chooses a new name for a port right, it can choose any name, other than `MACH_PORT_NULL` and `MACH_PORT_DEAD`, which is not currently being used for a port right or dead name. It might choose a name which at some previous time denoted a port right, but is currently unused.

4.2.4 Memory

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

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

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

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

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

Unaligned addresses and region sizes that are not page multiples are legal. A received message can also contain memory with unaligned addresses and funny sizes. In the general case, the first and last pages in the new memory region in the receiver do not contain only data from the sender, but are partly zero.[2] The received address points to the start of the data in the first page. This possibility doesn't complicate deallocation, because `vm_deallocate` does the right thing, rounding the start address down and the end address up to deallocate all arrived pages.

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

Out-of-line memory can carry port rights.

4.2.5 Message Send

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

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

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

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

`MACH_SEND_TIMEOUT`
: The timeout argument should specify a maximum time (in milliseconds) for the call to block before giving up.[3] If the message can't be queued before the timeout interval elapses, then the call returns `MACH_SEND_TIMED_OUT`. A zero timeout is legitimate.

`MACH_SEND_NOTIFY`
: The notify argument should specify a receive right for a notify port. If the send were to block, then instead the message is queued, `MACH_SEND_WILL_NOTIFY` is returned, and a msg-accepted notification is requested. If `MACH_SEND_TIMEOUT`

---

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

[3] If MACH_SEND_TIMEOUT is used without MACH_SEND_INTERRUPT, then the timeout duration might not be accurate. When the call is interrupted and automatically retried, the original timeout is used. If interrupts occur frequently enough, the timeout interval might never expire.

is also specified, then `MACH_SEND_NOTIFY` doesn't take effect until the timeout interval elapses.

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

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

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

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

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

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

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

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

`MACH_SEND_INVALID_DATA`
: The supplied message buffer was not readable.

`MACH_SEND_INVALID_HEADER`
: The `msgh_bits` value was invalid.

`MACH_SEND_INVALID_DEST`
: The `msgh_remote_port` value was invalid.

`MACH_SEND_INVALID_REPLY`
: The `msgh_local_port` value was invalid.

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

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

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

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

MACH_SEND_INVALID_TYPE
: A type descriptor was invalid.

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

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

MACH_SEND_TIMED_OUT
: The timeout interval expired.

MACH_SEND_INTERRUPTED
: A software interrupt occurred.

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

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

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

These return codes imply that the message was queued:

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

MACH_MSG_SUCCESS
: The message was queued.

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

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

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

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

4.2.6 Message Receive

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

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

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

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

The destination and reply ports are reversed in a received message header. The msgh_local_port field names the destination port, from which the message was received, and the msgh_remote_port field names the reply port right. The bits in msgh_bits are also reversed. The MACH_MSGH_BITS_LOCAL bits have the value MACH_MSG_TYPE_PORT_SEND if the message was sent to a send right, and the value MACH_MSG_TYPE_PORT_SEND_ONCE if was sent to a send-once right. The MACH_MSGH_BITS_REMOTE bits describe the reply port right.

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

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

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

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

`MACH_RCV_TIMEOUT`
: The timeout argument should specify a maximum time (in milliseconds) for the call to block before giving up.[4] If no message arrives before the timeout interval elapses, then the call returns `MACH_RCV_TIMED_OUT`. A zero timeout is legitimate.

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

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

`MACH_RCV_LARGE`
: If the message is larger than `rcv_size`, then the message remains queued instead of being destroyed. The call returns `MACH_RCV_TOO_LARGE` and the actual size of the message is returned in the `msgh_size` field of the message header.

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

`MACH_RCV_INVALID_NAME`
: The specified `rcv_name` was invalid.

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

`MACH_RCV_TIMED_OUT`
: The timeout interval expired.

`MACH_RCV_INTERRUPTED`
: A software interrupt occurred.

`MACH_RCV_PORT_DIED`
: The caller lost the rights specified by `rcv_name`.

[4] If MACH_RCV_TIMEOUT is used without MACH_RCV_INTERRUPT, then the timeout duration might not be accurate. When the call is interrupted and automatically retried, the original timeout is used. If interrupts occur frequently enough, the timeout interval might never expire.

MACH_RCV_PORT_CHANGED
: rcv_name specified a receive right which was moved into a port set during the call.

MACH_RCV_TOO_LARGE
: When using MACH_RCV_LARGE, and the message was larger than rcv_size. The message is left queued, and its actual size is returned in the msgh_size field of the message buffer.

These return codes imply that a message was dequeued and destroyed:

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

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

MACH_RCV_TOO_LARGE
: When not using MACH_RCV_LARGE, a message larger than rcv_size was dequeued and destroyed.

In these situations, when a message is dequeued and then destroyed, the reply port and all port rights and memory in the message body are destroyed. However, the caller receives the message's header, with all fields correct, including the destination port but excepting the reply port, which is MACH_PORT_NULL.

These return codes imply that a message was received:

MACH_RCV_BODY_ERROR
: A resource shortage prevented the reception of a port right or out-of-line memory region in the message body. The message header, including the reply port, is correct. The kernel attempts to transfer all port rights and memory regions in the body, and only destroys those that can't be transferred.

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

MACH_MSG_SUCCESS
: A message was received.

Resource shortages can occur after a message is dequeued, while transferring port rights and out-of-line memory regions to the receiving task. The mach_msg call returns MACH_RCV_HEADER_ERROR or MACH_RCV_BODY_ERROR in this situation. These return codes always carry extra bits (bitwise-ored) that indicate the nature of the resource shortage:

MACH_MSG_IPC_SPACE
: There was no room in the task's IPC name space for another port name.

MACH_MSG_VM_SPACE
: There was no room in the task's VM address space for an out-of-line memory region.

`MACH_MSG_IPC_KERNEL`
: A kernel resource shortage prevented the reception of a port right.

`MACH_MSG_VM_KERNEL`
: A kernel resource shortage prevented the reception of an out-of-line memory region.

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

4.2.7 Atomicity

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

For example, consider sending a message with the destination port specified as `MACH_MSG_TYPE_MOVE_SEND` and the reply port specified as `MACH_MSG_TYPE_COPY_SEND`. The same send right, with one user-reference, is supplied for both the `msgh_remote_port` and `msgh_local_port` fields. Because `mach_msg` processes the message header atomically, this succeeds. If `msgh_remote_port` were processed before `msgh_local_port`, then `mach_msg` would return `MACH_SEND_INVALID_REPLY` in this situation.

On the other hand, suppose the destination and reply port are both specified as `MACH_MSG_TYPE_MOVE_SEND`, and again the same send right with one user-reference is supplied for both. Now the send operation fails, but because it processes the header atomically, mach_msg can return either `MACH_SEND_INVALID_DEST` or `MACH_SEND_INVALID_REPLY`.

For example, consider receiving a message at the same time another thread is deallocating the destination receive right. Suppose the reply port field carries a send right for the destination port. If the deallocation happens before the dequeuing, then the receiver gets `MACH_RCV_PORT_DIED`. If the deallocation happens after the receive, then the `msgh_local_port` and the `msgh_remote_port` fields both specify the same right, which becomes a dead name when the receive right is deallocated. If the deallocation happens between the dequeue and the receive, then the `msgh_local_port` and `msgh_remote_port` fields both specify `MACH_PORT_DEAD`. Because the header is processed atomically, it is not possible for just one of the two fields to hold `MACH_PORT_DEAD`.

The `MACH_RCV_NOTIFY` option provides a more likely example. Suppose a message carrying a send-once right reply port is received with `MACH_RCV_NOTIFY` at the same time the reply port is destroyed. If the reply port is destroyed first, then `msgh_remote_port` specifies `MACH_PORT_DEAD` and the kernel does not generate a dead-name notification. If the reply port is destroyed after it is received, then `msgh_remote_port` specifies a dead name for which the kernel generates a dead-name notification. It is not possible to receive the reply port right and have it turn into a dead name before the dead-name notification is requested; as part of the message header the reply port is received atomically.

4.3 Port Manipulation Interface

This section describes the interface to create, destroy and manipulate ports, port rights and port sets.

`ipc_space_t` [Data type]

This is a `task_t` (and as such a `mach_port_t`), which holds a port name associated with a port that represents an IPC space in the kernel. An IPC space is used by the kernel to manage the port names and rights available to a task. The IPC space doesn't get a port name of its own. Instead the port name of the task containing the IPC space is used to name the IPC space of the task (as is indicated by the fact that the type of `ipc_space_t` is actually `task_t`).

The IPC spaces of tasks are the only ones accessible outside of the kernel.

4.3.1 Port Creation

`kern_return_t mach_port_allocate` (*ipc_space_t* `task`, *mach_port_right_t* `right`, *mach_port_t* `*name`) [Function]

The `mach_port_allocate` function creates a new right in the specified task. The new right's name is returned in *name*, which may be any name that wasn't in use.

The *right* argument takes the following values:

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

  task does not hold send rights for the new port, only the receive right. `mach_port_insert_right` and `mach_port_extract_right` can be used to convert the receive right into a combined send/receive right.

`MACH_PORT_RIGHT_PORT_SET`
: `mach_port_allocate` creates a port set. The new port set has no members.

`MACH_PORT_RIGHT_DEAD_NAME`
: `mach_port_allocate` creates a dead name. The new dead name has one user reference.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_VALUE` if *right* was invalid, `KERN_NO_SPACE` if there was no room in *task*'s IPC name space for another right and `KERN_RESOURCE_SHORTAGE` if the kernel ran out of memory.

The `mach_port_allocate` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`mach_port_t mach_reply_port ()` [Function]

The `mach_reply_port` system call creates a reply port in the calling task.

`mach_reply_port` creates a port, giving the calling task the receive right for the port. The call returns the name of the new receive right.

This is very much like creating a receive right with the `mach_port_allocate` call, with two differences. First, `mach_reply_port` is a system call and not an RPC (which requires a reply port). Second, the port created by `mach_reply_port` may be optimized for use as a reply port.

The function returns `MACH_PORT_NULL` if a resource shortage prevented the creation of the receive right.

`kern_return_t mach_port_allocate_name` (*ipc_space_t* `task`, *mach_port_right_t* `right`, *mach_port_t* `name`) [Function]

The function `mach_port_allocate_name` creates a new right in the specified task, with a specified name for the new right. *name* must not already be in use for some right, and it can't be the reserved values `MACH_PORT_NULL` and `MACH_PORT_DEAD`.

The *right* argument takes the following values:

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

: *task* does not hold send rights for the new port, only the receive right. `mach_port_insert_right` and `mach_port_extract_right` can be used to convert the receive right into a combined send/receive right.

`MACH_PORT_RIGHT_PORT_SET`
: `mach_port_allocate_name` creates a port set. The new port set has no members.

`MACH_PORT_RIGHT_DEAD_NAME`
: `mach_port_allocate_name` creates a new dead name. The new dead name has one user reference.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_VALUE` if *right* was invalid or *name* was `MACH_PORT_NULL` or `MACH_PORT_DEAD`, `KERN_NAME_EXISTS` if *name* was already in use for a port right and `KERN_RESOURCE_SHORTAGE` if the kernel ran out of memory.

The `mach_port_allocate_name` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

4.3.2 Port Destruction

`kern_return_t mach_port_deallocate` (*ipc_space_t* **task**, *mach_port_t* **name**) [Function]

The function `mach_port_deallocate` releases a user reference for a right in *task*'s IPC name space. It allows a task to release a user reference for a send or send-once right without failing if the port has died and the right is now actually a dead name.

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right and `KERN_INVALID_RIGHT` if *name* denoted an invalid right.

The `mach_port_deallocate` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t mach_port_destroy` (*ipc_space_t* **task**, *mach_port_t* **name**) [Function]

The function `mach_port_destroy` deallocates all rights denoted by a name. The name becomes immediately available for reuse.

For most purposes, `mach_port_mod_refs` and `mach_port_deallocate` are preferable.

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

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

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

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right.

The `mach_port_destroy` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

4.3.3 Port Names

`kern_return_t mach_port_names` (*ipc_space_t* `task`, *mach_port_array_t* `*names`, *mach_msg_type_number_t* `*ncount`, *mach_port_type_array_t* `*types`, *mach_msg_type_number_t* `*tcount`) [Function]

The function `mach_port_names` returns information about *task*'s port name space. For each name, it also returns what type of rights *task* holds. (The same information returned by `mach_port_type`.) *names* and *types* are arrays that are automatically allocated when the reply message is received. The user should `vm_deallocate` them when the data is no longer needed.

`mach_port_names` will return in *names* the names of the ports, port sets, and dead names in the task's port name space, in no particular order and in *ncount* the number of names returned. It will return in *types* the type of each corresponding name, which indicates what kind of rights the task holds with that name. *tcount* should be the same as *ncount*.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_RESOURCE_SHORTAGE` if the kernel ran out of memory.

The `mach_port_names` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t mach_port_type` (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_type_t* `*ptype`) [Function]

The function `mach_port_type` returns information about *task*'s rights for a specific name in its port name space. The returned *ptype* is a bitmask indicating what rights *task* holds for the port, port set or dead name. The bitmask is composed of the following bits:

`MACH_PORT_TYPE_SEND`
: The name denotes a send right.

`MACH_PORT_TYPE_RECEIVE`
: The name denotes a receive right.

`MACH_PORT_TYPE_SEND_ONCE`
: The name denotes a send-once right.

`MACH_PORT_TYPE_PORT_SET`
: The name denotes a port set.

`MACH_PORT_TYPE_DEAD_NAME`
: The name is a dead name.

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

`MACH_PORT_TYPE_MAREQUEST`
: A msg-accepted request for the right is pending.

`MACH_PORT_TYPE_COMPAT`
: The port right was created in the compatibility mode.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid and `KERN_INVALID_NAME` if *name* did not denote a right.

The `mach_port_type` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t` **`mach_port_rename`** (*ipc_space_t* `task`, *mach_port_t* `old_name`, *mach_port_t* `new_name`) [Function]

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *old_name* did not denote a right, `KERN_INVALID_VALUE` if *new_name* was `MACH_PORT_NULL` or `MACH_PORT_DEAD`, `KERN_NAME_EXISTS` if `new_name` already denoted a right and `KERN_RESOURCE_SHORTAGE` if the kernel ran out of memory.

The `mach_port_rename` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

4.3.4 Port Rights

`kern_return_t` **`mach_port_get_refs`** (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_right_t* `right`, *mach_port_urefs_t* `*refs`) [Function]

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

The *right* argument takes the following values:

- `MACH_PORT_RIGHT_SEND`
- `MACH_PORT_RIGHT_RECEIVE`
- `MACH_PORT_RIGHT_SEND_ONCE`
- `MACH_PORT_RIGHT_PORT_SET`
- `MACH_PORT_RIGHT_DEAD_NAME`

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_VALUE` if *right* was invalid and `KERN_INVALID_NAME` if *name* did not denote a right.

The `mach_port_get_refs` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t mach_port_mod_refs` (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_right_t* `right`, *mach_port_delta_t* `delta`) [Function]

The function `mach_port_mod_refs` requests that the number of user references a task has for a right be changed. This results in the right being destroyed, if the number of user references is changed to zero. The task holding the right is *task*, *name* should denote the specified right. *right* denotes the type of right being modified. *delta* is the signed change to the number of user references.

The *right* argument takes the following values:

- `MACH_PORT_RIGHT_SEND`
- `MACH_PORT_RIGHT_RECEIVE`
- `MACH_PORT_RIGHT_SEND_ONCE`
- `MACH_PORT_RIGHT_PORT_SET`
- `MACH_PORT_RIGHT_DEAD_NAME`

The number of user references for the right is changed by the amount *delta*, subject to the following restrictions: port sets, receive rights, and send-once rights may only have one user reference. The resulting number of user references can't be negative. If the resulting number of user references is zero, the effect is to deallocate the right. For dead names and send rights, there is an implementation-defined maximum number of user references.

If the call destroys the right, then the effect is as described for `mach_port_destroy`, with the exception that `mach_port_destroy` simultaneously destroys all the rights denoted by a name, while `mach_port_mod_refs` can only destroy one right. The name will be available for reuse if it only denoted the one right.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_VALUE` if *right* was invalid or the user-reference count would become negative, `KERN_INVALID_NAME` if *name* did not denote a right, `KERN_INVALID_RIGHT` if *name* denoted a right, but not the specified right and `KERN_UREFS_OVERFLOW` if the user-reference count would overflow.

The `mach_port_mod_refs` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

4.3.5 Ports and other Tasks

`kern_return_t mach_port_insert_right` (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_t* `right`, *mach_msg_type_name_t* `right_type`) [Function]

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

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

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

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

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

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_VALUE` if *right* was not a port right or *name* was `MACH_PORT_NULL` or `MACH_PORT_DEAD`, `KERN_NAME_EXISTS` if *name* already denoted a right, `KERN_INVALID_CAPABILITY` if *right* was `MACH_PORT_NULL` or `MACH_PORT_DEAD` `KERN_RIGHT_EXISTS` if *task* already had rights for the port, with a different name, `KERN_UREFS_OVERFLOW` if the user-reference count would overflow and `KERN_RESOURCE_SHORTAGE` if the kernel ran out of memory.

The `mach_port_insert_right` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t mach_port_extract_right` (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_msg_type_name_t* `desired_type`, *mach_port_t* `*right`, *mach_msg_type_name_t* `*acquired_type`) [Function]

The function *mach_port_extract_right* extracts a port right from the target *task* and returns it to the caller as if the task sent the right voluntarily, using *desired_type* as the value of *msgt_name*. See Section 4.2.1 [Mach Message Call], page 14.

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right, `KERN_INVALID_RIGHT` if *name* denoted a right, but an invalid one, `KERN_INVALID_VALUE` if *desired_type* was invalid.

The `mach_port_extract_right` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

4.3.6 Receive Rights

`mach_port_seqno_t` [Data type]

The `mach_port_seqno_t` data type is an `unsigned int` which contains the sequence number of a port.

`mach_port_mscount_t` [Data type]

The `mach_port_mscount_t` data type is an `unsigned int` which contains the make-send count for a port.

`mach_port_msgcount_t` [Data type]

The `mach_port_msgcount_t` data type is an `unsigned int` which contains a number of messages.

`mach_port_rights_t` [Data type]

The `mach_port_rights_t` data type is an `unsigned int` which contains a number of rights for a port.

`mach_port_status_t` [Data type]

This structure contains some status information about a port, which can be queried with `mach_port_get_receive_status`. It has the following members:

`mach_port_t mps_pset`
: The containing port set.

`mach_port_seqno_t mps_seqno`
: The sequence number.

`mach_port_mscount_t mps_mscount`
: The make-send count.

`mach_port_msgcount_t mps_qlimit`
: The maximum number of messages in the queue.

`mach_port_msgcount_t mps_msgcount`
: The current number of messages in the queue.

`mach_port_rights_t mps_sorights`
: The number of send-once rights that exist.

`boolean_t mps_srights`
: `TRUE` if send rights exist.

`boolean_t mps_pdrequest`
: `TRUE` if port-deleted notification is requested.

`boolean_t mps_nsrequest`
: `TRUE` if no-senders notification is requested.

`kern_return_t mach_port_get_receive_status` (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_status_t* `*status`) [Function]

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right and `KERN_INVALID_RIGHT` if *name* denoted a right, but not a receive right.

The `mach_port_get_receive_status` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t` **mach_port_set_mscount** (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_mscount_t* `mscount`) [Function]

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right and `KERN_INVALID_RIGHT` if *name* denoted a right, but not a receive right.

The `mach_port_set_mscount` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t` **mach_port_set_qlimit** (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_msgcount_t* `qlimit`) [Function]

The function `mach_port_set_qlimit` changes the queue limit *task*'s receive right named *name* to *qlimit*. Valid values for *qlimit* are between zero and `MACH_PORT_QLIMIT_MAX`, inclusive.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right, `KERN_INVALID_RIGHT` if *name* denoted a right, but not a receive right and `KERN_INVALID_VALUE` if *qlimit* was invalid.

The `mach_port_set_qlimit` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t` **mach_port_set_seqno** (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_seqno_t* `seqno`) [Function]

The function `mach_port_set_seqno` changes the sequence number *task*'s receive right named *name* to *seqno*. All sequence number values are valid. The next message received from the port will be stamped with the specified sequence number.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right and `KERN_INVALID_RIGHT` if *name* denoted a right, but not a receive right.

The `mach_port_set_seqno` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

4.3.7 Port Sets

`kern_return_t` **mach_port_get_set_status** (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_port_array_t* `*members`, *mach_msg_type_number_t* `*count`) [Function]

The function `mach_port_get_set_status` returns the members of a port set. *members* is an array that is automatically allocated when the reply message is received. The user should `vm_deallocate` it when the data is no longer needed.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right, `KERN_INVALID_RIGHT` if *name* denoted a right, but not a receive right and `KERN_RESOURCE_SHORTAGE` if the kernel ran out of memory.

The `mach_port_get_set_status` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

`kern_return_t` **mach_port_move_member** (*ipc_space_t* `task`, *mach_port_t* `member`, *mach_port_t* `after`) [Function]

The function *mach_port_move_member* moves the receive right *member* into the port set *after*. If the receive right is already a member of another port set, it is removed from that set first (the whole operation is atomic). If the port set is `MACH_PORT_NULL`, then the receive right is not put into a port set, but removed from its current port set.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_NAME` if *member* or *after* did not denote a right, `KERN_INVALID_RIGHT` if *member* denoted a right, but not a receive right or *after* denoted a right, but not a port set, and `KERN_NOT_IN_SET` if *after* was `MACH_PORT_NULL`, but `member` wasn't currently in a port set.

The `mach_port_move_member` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

4.3.8 Request Notifications

`kern_return_t` **mach_port_request_notification** (*ipc_space_t* `task`, *mach_port_t* `name`, *mach_msg_id_t* `variant`, *mach_port_mscount_t* `sync`, *mach_port_t* `notify`, *mach_msg_type_name_t* `notify_type`, *mach_port_t* `*previous`) [Function]

The function `mach_port_request_notification` registers a request for a notification and supplies the send-once right *notify* to which the notification will be sent. The *notify_type* denotes the IPC type for the send-once right, which can be `MACH_MSG_TYPE_MAKE_SEND_ONCE` or `MACH_MSG_TYPE_MOVE_SEND_ONCE`. It is an atomic swap, returning the previously registered send-once right (or `MACH_PORT_NULL` for none) in *previous*. A previous notification request may be cancelled by providing `MACH_PORT_NULL` for *notify*.

The *variant* argument takes the following values:

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

`MACH_NOTIFY_DEAD_NAME`
: The call requests a dead-name notification. *name* specifies send, receive, or send-once rights for a port. If the port is destroyed (and the right remains, becoming a dead name), then a dead-name notification which carries the name of the right will be sent to the registered send-once right. If *notify* is not null and sync is non-zero, the name may specify a dead name, and a dead-name notification is immediately generated.

  Whenever a dead-name notification is generated, the user reference count of the dead name is incremented. For example, a send right with two user refs has a registered dead-name request. If the port is destroyed, the send right turns into a dead name with three user refs (instead of two), and a dead-name notification is generated.

  If the name is made available for reuse, perhaps because of `mach_port_destroy` or `mach_port_mod_refs`, or the name denotes a send-once right which has a message sent to it, then the registered send-once right is used to generate a port-deleted notification.

`MACH_NOTIFY_NO_SENDERS`
: The call requests a no-senders notification. *name* must specify a receive right. If *notify* is not null, and the receive right's make-send count is greater than or equal to the sync value, and it has no extant send rights, than an immediate no-senders notification is generated. Otherwise the notification is generated when the receive right next loses its last extant send right. In either case, any previously registered send-once right is returned.

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

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_TASK` if *task* was invalid, `KERN_INVALID_VALUE` if *variant* was invalid, `KERN_INVALID_NAME` if *name* did not denote a right, `KERN_INVALID_RIGHT` if *name* denoted an invalid right and `KERN_INVALID_CAPABILITY` if *notify* was invalid.

When using `MACH_NOTIFY_PORT_DESTROYED`, the function returns `KERN_INVALID_VALUE` if *sync* wasn't zero.

When using `MACH_NOTIFY_DEAD_NAME`, the function returns `KERN_RESOURCE_SHORTAGE` if the kernel ran out of memory, `KERN_INVALID_ARGUMENT` if *name* denotes a dead name, but *sync* is zero or *notify* is `MACH_PORT_NULL`, and `KERN_UREFS_OVERFLOW` if *name* denotes a dead name, but generating an immediate dead-name notification would overflow the name's user-reference count.

The `mach_port_request_notification` call is actually an RPC to *task*, normally a send right for a task port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

5 Virtual Memory Interface

`vm_task_t` [Data type]

This is a `task_t` (and as such a `mach_port_t`), which holds a port name associated with a port that represents a virtual memory map in the kernel. An virtual memory map is used by the kernel to manage the address space of a task. The virtual memory map doesn't get a port name of its own. Instead the port name of the task provided with the virtual memory is used to name the virtual memory map of the task (as is indicated by the fact that the type of `vm_task_t` is actually `task_t`).

The virtual memory maps of tasks are the only ones accessible outside of the kernel.

5.1 Memory Allocation

`kern_return_t vm_allocate` (*vm_task_t* `target_task`, *vm_address_t* `*address`, *vm_size_t* `size`, *boolean_t* `anywhere`) [Function]

The function `vm_allocate` allocates a region of virtual memory, placing it in the specified *task*'s address space.

The starting address is *address*. If the *anywhere* option is false, an attempt is made to allocate virtual memory starting at this virtual address. If this address is not at the beginning of a virtual page, it will be rounded down to one. If there is not enough space at this address, no memory will be allocated. If the *anywhere* option is true, the input value of this address will be ignored, and the space will be allocated wherever it is available. In either case, the address at which memory was actually allocated will be returned in *address*.

size is the number of bytes to allocate (rounded by the system in a machine dependent way to an integral number of virtual pages).

If *anywhere* is true, the kernel should find and allocate any region of the specified size, and return the address of the resulting region in address address, rounded to a virtual page boundary if there is sufficient space.

The physical memory is not actually allocated until the new virtual memory is referenced. By default, the kernel rounds all addresses down to the nearest page boundary and all memory sizes up to the nearest page size. The global variable `vm_page_size` contains the page size. `mach_task_self` returns the value of the current task port which should be used as the *target_task* argument in order to allocate memory in the caller's address space. For languages other than C, these values can be obtained by the calls `vm_statistics` and `mach_task_self`. Initially, the pages of allocated memory will be protected to allow all forms of access, and will be inherited in child tasks as a copy. Subsequent calls to `vm_protect` and `vm_inherit` may be used to change these properties. The allocated region is always zero-filled.

The function returns `KERN_SUCCESS` if the memory was successfully allocated, `KERN_INVALID_ADDRESS` if an invalid address was specified and `KERN_NO_SPACE` if there was not enough space left to satisfy the request.

5.2 Memory Deallocation

`kern_return_t vm_deallocate` (*vm_task_t* `target_task`, *vm_address_t* `address`, *vm_size_t* `size`) [Function]

`vm_deallocate` relinquishes access to a region of a *task*'s address space, causing further access to that memory to fail. This address range will be available for reallocation. *address* is the starting address, which will be rounded down to a page boundary. *size* is the number of bytes to deallocate, which will be rounded up to give a page boundary. Note, that because of the rounding to virtual page boundaries, more than *size* bytes may be deallocated. Use `vm_page_size` or `vm_statistics` to find out the current virtual page size.

This call may be used to deallocte memory that was passed to a task in a message (via out of line data). In that case, the rounding should cause no trouble, since the region of memory was allocated as a set of pages.

The `vm_deallocate` call affects only the task specified by the *target_task*. Other tasks which may have access to this memory may continue to reference it.

The function returns `KERN_SUCCESS` if the memory was successfully deallocated and `KERN_INVALID_ADDRESS` if an invalid or non-allocated address was specified.

5.3 Data Transfer

`kern_return_t vm_read` (*vm_task_t* `target_task`, *vm_address_t* `address`, *vm_size_t* `size`, *vm_offset_t* *`data`, *mach_msg_type_number_t* *`data_count`) [Function]

The function `vm_read` allows one task's virtual memory to be read by another task. The *target_task* is the task whose memory is to be read. *address* is the first address to be read and must be on a page boundary. *size* is the number of bytes of data to be read and must be an integral number of pages. *data* is the array of data copied from the given task, and *data_count* is the size of the data array in bytes (will be an integral number of pages).

Note that the data array is returned in a newly allocated region; the task reading the data should `vm_deallocate` this region when it is done with the data.

The function returns `KERN_SUCCESS` if the memory was successfully read, `KERN_INVALID_ADDRESS` if an invalid or non-allocated address was specified or there was not *size* bytes of data following the address, `KERN_INVALID_ARGUMENT` if the address does not start on a page boundary or the size is not an integral number of pages, `KERN_PROTECTION_FAILURE` if the address region in the target task is protected against reading and `KERN_NO_SPACE` if there was not enough room in the callers virtual memory to allocate space for the data to be returned.

`kern_return_t vm_write` (*vm_task_t* `target_task`, *vm_address_t* `address`, *vm_offset_t* `data`, *mach_msg_type_number_t* `data_count`) [Function]

The function `vm_write` allows a task to write to the vrtual memory of *target_task*. *address* is the starting address in task to be affected. *data* is an array of bytes to be written, and *data_count* the size of the *data* array.

The current implementation requires that *address*, *data* and *data_count* all be page-aligned. Otherwise, `KERN_INVALID_ARGUMENT` is returned.

The function returns `KERN_SUCCESS` if the memory was successfully written, `KERN_INVALID_ADDRESS` if an invalid or non-allocated address was specified or there was not *data_count* bytes of allocated memory starting at *address* and `KERN_PROTECTION_FAILURE` if the address region in the target task is protected against writing.

`kern_return_t vm_copy` (*vm_task_t* `target_task`, *vm_address_t* `source_address`, *vm_size_t* `count`, *vm_offset_t* `dest_address`) [Function]

The function `vm_copy` causes the source memory range to be copied to the destination address. The source and destination memory ranges may overlap. The destination address range must already be allocated and writable; the source range must be readable.

`vm_copy` is equivalent to `vm_read` followed by `vm_write`.

The current implementation requires that *address*, *data* and *data_count* all be page-aligned. Otherwise, `KERN_INVALID_ARGUMENT` is returned.

The function returns `KERN_SUCCESS` if the memory was successfully written, `KERN_INVALID_ADDRESS` if an invalid or non-allocated address was specified or there was insufficient memory allocated at one of the addresses and `KERN_PROTECTION_FAILURE` if the destination region was not writable or the source region was not readable.

5.4 Memory Attributes

`kern_return_t vm_region` (*vm_task_t* `target_task`, *vm_address_t* *`address`, *vm_size_t* *`size`, *vm_prot_t* *`protection`, *vm_prot_t* *`max_protection`, *vm_inherit_t* *`inheritance`, *boolean_t* *`shared`, *memory_object_name_t* *`object_name`, *vm_offset_t* *`offset`) [Function]

The function `vm_region` returns a description of the specified region of *target_task*'s virtual address space. `vm_region` begins at *address* and looks forward through memory until it comes to an allocated region. If address is within a region, then that region is used. Various bits of information about the region are returned. If *address* was not within a region, then *address* is set to the start of the first region which follows the incoming value. In this way an entire address space can be scanned.

The *size* returned is the size of the located region in bytes. *protection* is the current protection of the region, *max_protection* is the maximum allowable protection for this region. *inheritance* is the inheritance attribute for this region. *shared* tells if the region is shared or not. The port *object_name* identifies the memory object associated with this region, and *offset* is the offset into the pager object that this region begins at.

The function returns `KERN_SUCCESS` if the memory region was successfully located and the information returned and `KERN_NO_SPACE` if there is no region at or above *address* in the specified task.

`kern_return_t vm_protect` (*vm_task_t* `target_task`, *vm_address_t* `address`, *vm_size_t* `size`, *boolean_t* `set_maximum`, *vm_prot_t* `new_protection`) [Function]

The function `vm_protect` sets the virtual memory access privileges for a range of allocated addresses in *target_task*'s virtual address space. The protection argument describes a combination of read, write, and execute accesses that should be *permitted*.

address is the starting address, which will be rounded down to a page boundary. *size* is the size in bytes of the region for which protection is to change, and will be rounded up to give a page boundary. If *set_maximum* is set, make the protection change apply to the maximum protection associated with this address range; otherwise, the current protection on this range is changed. If the maximum protection is reduced below the current protection, both will be changed to reflect the new maximum. *new_protection* is the new protection value for this region; a set of: `VM_PROT_READ`, `VM_PROT_WRITE`, `VM_PROT_EXECUTE`.

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

The function returns `KERN_SUCCESS` if the memory was successfully protected, `KERN_INVALID_ADDRESS` if an invalid or non-allocated address was specified and `KERN_PROTECTION_FAILURE` if an attempt was made to increase the current or maximum protection beyond the existing maximum protection value.

`kern_return_t vm_inherit` (*vm_task_t* `target_task`, *vm_address_t* `address`, *vm_size_t* `size`, *vm_inherit_t* `new_inheritance`) [Function]

The function `vm_inherit` specifies how a region of *target_task*'s address space is to be passed to child tasks at the time of task creation. Inheritance is an attribute of virtual pages, so *address* to start from will be rounded down to a page boundary and *size*, the size in bytes of the region for wihch inheritance is to change, will be rounded up to give a page boundary. How this memory is to be inherited in child tasks is specified by *new_inheritance*. Inheritance is specified by using one of these following three values:

`VM_INHERIT_SHARE`
: Child tasks will share this memory with this task.

`VM_INHERIT_COPY`
: Child tasks will receive a copy of this region.

`VM_INHERIT_NONE`
: This region will be absent from child tasks.

Setting `vm_inherit` to `VM_INHERIT_SHARE` and forking a child task is the only way two Mach tasks can share physical memory. Remember that all the theads of a given task share all the same memory.

The function returns `KERN_SUCCESS` if the memory inheritance was successfully set and `KERN_INVALID_ADDRESS` if an invalid or non-allocated address was specified.

kern_return_t vm_wire (*host_priv_t* host_priv, *vm_task_t* target_task, *vm_address_t* address, *vm_size_t* size, *vm_prot_t* access) [Function]

The function vm_wire allows privileged applications to control memory pageability. *host_priv* is the privileged host port for the host on which *target_task* resides. *address* is the starting address, which will be rounded down to a page boundary. *size* is the size in bytes of the region for which protection is to change, and will be rounded up to give a page boundary. *access* specifies the types of accesses that must not cause page faults.

The semantics of a successful vm_wire operation are that memory in the specified range will not cause page faults for any accesses included in access. Data memory can be made non-pageable (wired) with a access argument of VM_PROT_READ | VM_PROT_WRITE. A special case is that VM_PROT_NONE makes the memory pageable.

The function returns KERN_SUCCESS if the call succeeded, KERN_INVALID_HOST if *host_priv* was not the privileged host port, KERN_INVALID_TASK if *task* was not a valid task, KERN_INVALID_VALUE if *access* specified an invalid access mode, KERN_FAILURE if some memory in the specified range is not present or has an inappropriate protection value, and KERN_INVALID_ARGUMENT if unwiring (*access* is VM_PROT_NONE) and the memory is not already wired.

The vm_wire call is actually an RPC to *host_priv*, normally a send right for a privileged host port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return mach_msg return codes.

kern_return_t vm_machine_attribute (*vm_task_t* task, *vm_address_t* address, *vm_size_t* size, *vm_prot_t* access, *vm_machine_attribute_t* attribute, *vm_machine_attribute_val_t* value) [Function]

The function vm_machine_attribute specifies machine-specific attributes for a VM mapping, such as cachability, migrability, replicability. This is used on machines that allow the user control over the cache (this is the case for MIPS architectures) or placement of memory pages as in NUMA architectures (Non-Uniform Memory Access time) such as the IBM ACE multiprocessor.

Machine-specific attributes can be consider additions to the machine-independent ones such as protection and inheritance, but they are not guaranteed to be supported by any given machine. Moreover, implementations of Mach on new architectures might find the need for new attribute types and or values besides the ones defined in the initial implementation.

The types currently defined are

MATTR_CACHE
: Controls caching of memory pages

MATTR_MIGRATE
: Controls migrability of memory pages

MATTR_REPLICATE
: Controls replication of memory pages

Corresponding values, and meaning of a specific call to `vm_machine_attribute`

`MATTR_VAL_ON`
: Enables the attribute. Being enabled is the default value for any applicable attribute.

`MATTR_VAL_OFF`
: Disables the attribute, making memory non-cached, or non-migratable, or non-replicatable.

`MATTR_VAL_GET`
: Returns the current value of the attribute for the memory segment. If the attribute does not apply uniformly to the given range the value returned applies to the initial portion of the segment only.

`MATTR_VAL_CACHE_FLUSH`
: Flush the memory pages from the Cache. The size value in this case might be meaningful even if not a multiple of the page size, depending on the implementation.

`MATTR_VAL_ICACHE_FLUSH`
: Same as above, applied to the Instruction Cache alone.

`MATTR_VAL_DCACHE_FLUSH`
: Same as above, applied to the Data Cache alone.

The function returns `KERN_SUCCESS` if call succeeded, and `KERN_INVALID_ARGUMENT` if *task* is not a task, or *address* and *size* do not define a valid address range in task, or *attribute* is not a valid attribute type, or it is not implemented, or *value* is not a permissible value for attribute.

5.5 Mapping Memory Objects

`kern_return_t` **`vm_map`** (*vm_task_t* `target_task`, *vm_address_t* `*address`, *vm_size_t* `size`, *vm_address_t* `mask`, *boolean_t* `anywhere`, *memory_object_t* `memory_object`, *vm_offset_t* `offset`, *boolean_t* `copy`, *vm_prot_t* `cur_protection`, *vm_prot_t* `max_protection`, *vm_inherit_t* `inheritance`) [Function]

The function `vm_map` maps a region of virtual memory at the specified address, for which data is to be supplied by the given memory object, starting at the given offset within that object. In addition to the arguments used in `vm_allocate`, the `vm_map` call allows the specification of an address alignment parameter, and of the initial protection and inheritance values.

If the memory object in question is not currently in use, the kernel will perform a `memory_object_init` call at this time. If the copy parameter is asserted, the specified region of the memory object will be copied to this address space; changes made to this object by other tasks will not be visible in this mapping, and changes made in this mapping will not be visible to others (or returned to the memory object).

The `vm_map` call returns once the mapping is established. Completion of the call does not require any action on the part of the memory manager.

Warning: Only memory objects that are provided by bona fide memory managers should be used in the `vm_map` call. A memory manager must implement the memory object interface described elsewhere in this manual. If other ports are used, a thread that accesses the mapped virtual memory may become permanently hung or may receive a memory exception.

target_task is the task to be affected. The starting address is *address*. If the *anywhere* option is used, this address is ignored. The address actually allocated will be returned in *address*. *size* is the number of bytes to allocate (rounded by the system in a machine dependent way). The alignment restriction is specified by *mask*. Bits asserted in this mask must not be asserted in the address returned. If *anywhere* is set, the kernel should find and allocate any region of the specified size, and return the address of the resulting region in *address*.

memory_object is the port that represents the memory object: used by user tasks in `vm_map`; used by the make requests for data or other management actions. If this port is `MEMORY_OBJECT_NULL`, then zero-filled memory is allocated instead. Within a memory object, *offset* specifes an offset in bytes. This must be page aligned. If *copy* is set, the range of the memory object should be copied to the target task, rather than mapped read-write.

The function returns `KERN_SUCCESS` if the object is mapped, `KERN_NO_SPACE` if no unused region of the task's virtual address space that meets the address, size, and alignment criteria could be found, and `KERN_INVALID_ARGUMENT` if an invalid argument was provided.

5.6 Memory Statistics

`vm_statistics_data_t` [Data type]

This structure is returned in *vm_stats* by the `vm_statistics` function and provides virtual memory statistics for the system. It has the following members:

`long pagesize`
: The page size in bytes.

`long free_count`
: The number of free pages.

`long active_count`
: The umber of active pages.

`long inactive_count`
: The number of inactive pages.

`long wire_count`
: The number of pages wired down.

`long zero_fill_count`
: The number of zero filled pages.

`long reactivations`
: The number of reactivated pages.

`long pageins`
: The number of pageins.

`long pageouts`
: The number of pageouts.

`long faults`
: The number of faults.

`long cow_faults`
: The number of copy-on-writes.

`long lookups`
: The number of object cache lookups.

`long hits`
: The number of object cache hits.

`kern_return_t` **vm_statistics** (*vm_task_t target_task*, *vm_statistics_data_t *vm_stats*) [Function]

The function `vm_statistics` returns the statistics about the kernel's use of virtual memory since the kernel was booted. `pagesize` can also be found as a global variable `vm_page_size` which is set at task initialization and remains constant for the life of the task.

6 External Memory Management

6.1 Memory Object Server

boolean_t **memory_object_server** (*msg_header_t* *in_msg, *msg_header_t* *out_msg) [Function]
boolean_t **memory_object_default_server** (*msg_header_t* *in_msg, *msg_header_t* *out_msg) [Function]
boolean_t **seqnos_memory_object_server** (*msg_header_t* *in_msg, *msg_header_t* *out_msg) [Function]
boolean_t **seqnos_memory_object_default_server** (*msg_header_t* *in_msg, *msg_header_t* *out_msg) [Function]

A memory manager is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel.

In order to isolate the memory manager from the specifics of message formatting, the remote procedure call generator produces a procedure, `memory_object_server`, to handle a received message. This function does all necessary argument handling, and actually calls one of the following functions: `memory_object_init`, `memory_object_data_write`, `memory_object_data_return`, `memory_object_data_request`, `memory_object_data_unlock`, `memory_object_lock_completed`, `memory_object_copy`, `memory_object_terminate`. The **default memory manager** may get two additional requests from the kernel: `memory_object_create` and `memory_object_data_initialize`. The remote procedure call generator produces a procedure `memory_object_default_server` to handle those functions specific to the default memory manager.

The `seqnos_memory_object_server` and `seqnos_memory_object_default_server` differ from `memory_object_server` and `memory_object_default_server` in that they supply message sequence numbers to the server interfaces. They call the `seqnos_memory_object_*` functions, which complement the `memory_object_*` set of functions.

The return value from the `memory_object_server` function indicates that the message was appropriate to the memory management interface (returning `TRUE`), or that it could not handle this message (returning `FALSE`).

The *in_msg* argument is the message that has been received from the kernel. The *out_msg* is a reply message, but this is not used for this server.

The function returns `TRUE` to indicate that the message in question was applicable to this interface, and that the appropriate routine was called to interpret the message. It returns `FALSE` to indicate that the message did not apply to this interface, and that no other action was taken.

6.2 Memory Object Creation

`kern_return_t memory_object_init` [Function]
(*memory_object_t* `memory_object`, *memory_object_control_t* `memory_control`, *memory_object_name_t* `memory_object_name`, *vm_size_t* `memory_object_page_size`)

`kern_return_t seqnos_memory_object_init` [Function]
(*memory_object_t* `memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_control_t* `memory_control`, *memory_object_name_t* `memory_object_name`, *vm_size_t* `memory_object_page_size`)

The function `memory_object_init` serves as a notification that the kernel has been asked to map the given memory object into a task's virtual address space. Additionally, it provides a port on which the memory manager may issue cache management requests, and a port which the kernel will use to name this data region. In the event that different each will perform a `memory_object_init` call with new request and name ports. The virtual page size that is used by the calling kernel is included for planning purposes.

When the memory manager is prepared to accept requests for data for this object, it must call `memory_object_ready` with the attribute. Otherwise the kernel will not process requests on this object. To reject all mappings of this object, the memory manager may use `memory_object_destroy`.

The argument *memory_object* is the port that represents the memory object data, as supplied to the kernel in a `vm_map` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *memory_object_name* is a port used by the kernel to refer to the memory object data in reponse to `vm_region` calls. `memory_object_page_size` is the page size to be used by this kernel. All data sizes in calls involving this kernel must be an integral multiple of the page size. Note that different kernels, indicated by a different `memory_control`, may have different page sizes.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

`kern_return_t memory_object_ready` [Function]
(*memory_object_control_t* `memory_control`, *boolean_t* `may_cache_object`, *memory_object_copy_strategy_t* `copy_strategy`)

The function `memory_object_ready` informs the kernel that the memory manager is ready to receive data or unlock requests on behalf of the clients. The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. If *may_cache_object* is set, the kernel may keep data associated with this memory object, even after virtual memory references to it are gone.

copy_strategy tells how the kernel should copy regions of the associated memory object. There are three possible caching strategies: `MEMORY_OBJECT_COPY_NONE` which specifies that nothing special should be done when data in the object is copied; `MEMORY_OBJECT_COPY_CALL` which specifies that the memory manager should be notified via a `memory_object_copy` call before any part of the object is copied; and `MEMORY_OBJECT_COPY_DELAY` which guarantees that the memory manager does not externally modify the data so that the kernel can use its normal copy-on-write algorithms. `MEMORY_OBJECT_COPY_DELAY` is the strategy most commonly used.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

6.3 Memory Object Termination

`kern_return_t memory_object_terminate` [Function]
(*memory_object_t* `memory_object`, *memory_object_control_t* `memory_control`, *memory_object_name_t* `memory_object_name`)

`kern_return_t seqnos_memory_object_terminate` [Function]
(*memory_object_t* `memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_control_t* `memory_control`, *memory_object_name_t* `memory_object_name`)

The function `memory_object_terminate` indicates that the kernel has completed its use of the given memory object. All rights to the memory object control and name ports are included, so that the memory manager can destroy them (using `mach_port_deallocate`) after doing appropriate bookkeeping. The kernel will terminate a memory object only after all address space mappings of that memory object have been deallocated, or upon explicit request by the memory manager.

The argument *memory_object* is the port that represents the memory object data, as supplied to the kernel in a `vm_map` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *memory_object_name* is a port used by the kernel to refer to the memory object data in reponse to `vm_region` calls.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

`kern_return_t memory_object_destroy` [Function]
(*memory_object_control_t* `memory_control`, *kern_return_t* `reason`)

The function `memory_object_destroy` tells the kernel to shut down the memory object. As a result of this call the kernel will no longer support paging activity or any `memory_object` calls on this object, and all rights to the memory object port, the memory control port and the memory name port will be returned to the memory manager in a memory_object_terminate call. If the memory manager is concerned that any modified cached data be returned to it before the object is terminated, it should call `memory_object_lock_request` with *should_flush* set and a lock value of `VM_PROT_WRITE` before making this call.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. *reason* is an error code indicating why the object must be destroyed.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

6.4 Memory Objects and Data

`kern_return_t memory_object_data_return` [Function]
(*memory_object_t* `memory_object`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `data`, *vm_size_t* `data_count`, *boolean_t* `dirty`, *boolean_t* `kernel_copy`)

`kern_return_t seqnos_memory_object_data_return` [Function]
(*memory_object_t* `memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `data`, *vm_size_t* `data_count`, *boolean_t* `dirty`, *boolean_t* `kernel_copy`)

The function `memory_object_data_return` provides the memory manager with data that has been modified while cached in physical memory. Once the memory manager no longer needs this data (e.g., it has been written to another storage medium), it should be deallocated using `vm_deallocate`.

The argument *memory_object* is the port that represents the memory object data, as supplied to the kernel in a `vm_map` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *offset* is the offset within a memory object to which this call refers. This will be page aligned. *data* is the data which has been modified while cached in physical memory. *data_count* is the amount of data to be written, in bytes. This will be an integral number of memory object pages.

The kernel will also use this call to return precious pages. If an unmodified precious age is returned, *dirty* is set to `FALSE`, otherwise it is `TRUE`. If *kernel_copy* is `TRUE`, the kernel kept a copy of the page. Precious data remains precious if the kernel keeps a copy. The indication that the kernel kept a copy is only a hint if the data is not precious; the cleaned copy may be discarded without further notifying the manager.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

`kern_return_t memory_object_data_request` [Function]
(*memory_object_t* `memory_object`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `length`, *vm_prot_t* `desired_access`)

`kern_return_t seqnos_memory_object_data_request` [Function]
(*memory_object_t* `memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `length`, *vm_prot_t* `desired_access`)

The function `memory_object_data_request` is a request for data from the specified memory object, for at least the access specified. The memory manager is expected to return at least the specified data, with as much access as it can allow, using `memory_object_data_supply`. If the memory manager is unable to provide the data (for example, because of a hardware error), it may use the `memory_object_data_error` call. The `memory_object_data_unavailable` call may be used to tell the kernel to supply zero-filled memory for this region.

The argument *memory_object* is the port that represents the memory object data, as supplied to the kernel in a `vm_map` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *offset* is the offset within a memory object to which this call refers. This will be page aligned. *length* is the number of bytes of data, starting at *offset*, to which this call refers. This will be an integral number of memory object pages. *desired_access* is a protection value describing the memory access modes which must be permitted on the specified cached data. One or more of: `VM_PROT_READ`, `VM_PROT_WRITE` or `VM_PROT_EXECUTE`.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

`kern_return_t memory_object_data_supply` [Function]
(*memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `data`, *vm_size_t* `data_count`, *vm_prot_t* `lock_value`, *boolean_t* `precious`, *mach_port_t* `reply`)

The function `memory_object_data_supply` supplies the kernel with data for the specified memory object. Ordinarily, memory managers should only provide data in reponse to `memory_object_data_request` calls from the kernel (but they may provide data in advance as desired). When data already held by this kernel is provided again, the new data is ignored. The kernel may not provide any data (or protection) consistency among pages with different virtual page alignments within the same object.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. *offset* is an offset within a memory object in bytes. This must be page aligned. *data* is the data that is being provided to the kernel. This is a pointer to the data. *data_count* is the amount of data to be provided. Only whole virtual pages of data can be accepted; partial pages will be discarded.

lock_value is a protection value indicating those forms of access that should **not** be permitted to the specified cached data. The lock values must be one or more of the set: `VM_PROT_NONE`, `VM_PROT_READ`, `VM_PROT_WRITE`, `VM_PROT_EXECUTE` and `VM_PROT_ALL` as defined in '`mach/vm_prot.h`'.

If *precious* is `FALSE`, the kernel treats the data as a temporary and may throw it away if it hasn't been changed. If the *precious* value is `TRUE`, the kernel treats its copy as a data repository and promises to return it to the manager; the manager may tell the kernel to throw it away instead by flushing and not cleaning the data (see `memory_object_lock_request`).

If *reply_to* is not `MACH_PORT_NULL`, the kernel will send a completion message to the provided port (see `memory_object_supply_completed`).

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

`kern_return_t memory_object_supply_completed` *(memory_object_t* `memory_object`*, memory_object_control_t* `memory_control`*, vm_offset_t* `offset`*, vm_size_t* `length`*, kern_return_t* `result`*, vm_offset_t* `error_offset`*)* [Function]

`kern_return_t seqnos_memory_object_supply_completed` *(memory_object_t* `memory_object`*, mach_port_seqno_t* `seqno`*, memory_object_control_t* `memory_control`*, vm_offset_t* `offset`*, vm_size_t* `length`*, kern_return_t* `result`*, vm_offset_t* `error_offset`*)* [Function]

The function `memory_object_supply_completed` indicates that a previous `memory_object_data_supply` has been completed. Note that this call is made on whatever port was specified in the `memory_object_data_supply` call; that port need not be the memory object port itself. No reply is expected after this call.

The argument *memory_object* is the port that represents the memory object data, as supplied to the kernel in a `vm_map` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *offset* is the offset within a memory object to which this call refers. *length* is the length of the data covered by the lock request. The *result* parameter indicates what happened during the supply. If it is not `KERN_SUCCESS`, then *error_offset* identifies the first offset at which a problem occurred. The pagein operation stopped at this point. Note that the only failures reported by this mechanism are `KERN_MEMORY_PRESENT`. All other failures (invalid argument, error on pagein of supplied data in manager's address space) cause the entire operation to fail.

`kern_return_t memory_object_data_error` *(memory_object_control_t* `memory_control`*, vm_offset_t* `offset`*, vm_size_t* `size`*, kern_return_t* `reason`*)* [Function]

The function `memory_object_data_error` indicates that the memory manager cannot return the data requested for the given region, specifying a reason for the error. This is typically used when a hardware error is encountered.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. *offset* is an offset within a memory object in bytes. This must be page aligned. *data* is the data that is being provided to the kernel. This is a pointer to the data. *size* is the amount of cached data (starting at *offset*) to be handled. This must be an integral number of the memory object page size. *reason* is an error code indicating what type of error occured.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

`kern_return_t memory_object_data_unavailable` *(memory_object_control_t* `memory_control`*, vm_offset_t* `offset`*, vm_size_t* `size`*, kern_return_t* `reason`*)* [Function]

The function `memory_object_data_unavailable` indicates that the memory object does not have data for the given region and that the kernel should provide the data for this range. The memory manager may use this call in three different situations.

1. The object was created by `memory_object_create` and the kernel has not yet provided data for this range (either via a `memory_object_data_initialize`, `memory_object_data_write` or a `memory_object_data_return` for the object.
2. The object was created by an `memory_object_data_copy` and the kernel should copy this region from the original memory object.
3. The object is a normal user-created memory object and the kernel should supply unlocked zero-filled pages for the range.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. *offset* is an offset within a memory object, in bytes. This must be page aligned. *size* is the amount of cached data (starting at *offset*) to be handled. This must be an integral number of the memory object page size.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

`kern_return_t memory_object_copy` [Function]
(*memory_object_t* `old_memory_object`, *memory_object_control_t* `old_memory_control`, *vm_offset_t* `offset`, *vm_size_t* `length`, *memory_object_t* `new_memory_object`)

`kern_return_t seqnos_memory_object_copy` [Function]
(*memory_object_t* `old_memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_control_t* `old_memory_control`, *vm_offset_t* `offset`, *vm_size_t* `length`, *memory_object_t* `new_memory_object`)

The function `memory_object_copy` indicates that a copy has been made of the specified range of the given original memory object. This call includes only the new memory object itself; a `memory_object_init` call will be made on the new memory object after the currently cached pages of the original object are prepared. After the memory manager receives the init call, it must reply with the `memory_object_ready` call to assert the `"ready"` attribute. The kernel will use the new memory object, control and name ports to refer to the new copy.

This call is made when the original memory object had the caching parameter set to `MEMORY_OBJECT_COPY_CALL` and a user of the object has asked the kernel to copy it.

Cached pages from the original memory object at the time of the copy operation are handled as follows: Readable pages may be silently copied to the new memory object (with all access permissions). Pages not copied are locked to prevent write access.

The new memory object is **temporary**, meaning that the memory manager should not change its contents or allow the memory object to be mapped in another client. The memory manager may use the `memory_object_data_unavailable` call to indicate that the appropriate pages of the original memory object may be used to fulfill the data request.

The argument *old_memory_object* is the port that represents the old memory object data. *old_memory_control* is the kernel port for the old object. *offset* is the offset within a memory object to which this call refers. This will be page aligned. *length* is the number of bytes of data, starting at *offset*, to which this call refers. This will be an integral number of memory object pages. *new_memory_object* is a new memory object created by the kernel; see synopsis for further description. Note that all port rights (including receive rights) are included for the new memory object.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

The remaining interfaces in this section are obsolet.

`kern_return_t` **`memory_object_data_write`** (*memory_object_t* `memory_object`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `data`, *vm_size_t* `data_count`) [Function]

`kern_return_t` **`seqnos_memory_object_data_write`** (*memory_object_t* `memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `data`, *vm_size_t* `data_count`) [Function]

The function `memory_object_data_write` provides the memory manager with data that has been modified while cached in physical memory. It is the old form of `memory_object_data_return`. Once the memory manager no longer needs this data (e.g., it has been written to another storage medium), it should be deallocated using `vm_deallocate`.

The argument *memory_object* is the port that represents the memory object data, as supplied to the kernel in a `vm_map` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *offset* is the offset within a memory object to which this call refers. This will be page aligned. *data* is the data which has been modified while cached in physical memory. *data_count* is the amount of data to be written, in bytes. This will be an integral number of memory object pages.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

`kern_return_t` **`memory_object_data_provided`** (*memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `data`, *vm_size_t* `data_count`, *vm_prot_t* `lock_value`) [Function]

The function `memory_object_data_provided` supplies the kernel with data for the specified memory object. It is the old form of `memory_object_data_supply`. Ordinarily, memory managers should only provide data in reponse to `memory_object_data_request` calls from the kernel. The *lock_value* specifies what type of access will not be allowed to the data range. The lock values must be one or more of the set: `VM_PROT_NONE`, `VM_PROT_READ`, `VM_PROT_WRITE`, `VM_PROT_EXECUTE` and `VM_PROT_ALL` as defined in '`mach/vm_prot.h`'.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. *offset* is an offset within a memory object in bytes. This must be page aligned. *data* is the data that is being provided to the kernel. This is a pointer to the data. *data_count* is the amount of data to be provided. This must be an integral number of memory object pages. *lock_value* is a protection value indicating those forms of access that should **not** be permitted to the specified cached data.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

6.5 Memory Object Locking

`kern_return_t memory_object_lock_request` [Function]
(*memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_size_t* `size`, *memory_object_return_t* `should_clean`, *boolean_t* `should_flush`, *vm_prot_t* `lock_value`, *mach_port_t* `reply_to`)

The function `memory_object_lock_request` allows a memory manager to make cache management requests. As specified in arguments to the call, the kernel will:

- clean (i.e., write back using `memory_object_data_supply` or `memory_object_data_write`) any cached data which has been modified since the last time it was written
- flush (i.e., remove any uses of) that data from memory
- lock (i.e., prohibit the specified uses of) the cached data

Locks applied to cached data are not cumulative; new lock values override previous ones. Thus, data may also be unlocked using this primitive. The lock values must be one or more of the following values: `VM_PROT_NONE`, `VM_PROT_READ`, `VM_PROT_WRITE`, `VM_PROT_EXECUTE` and `VM_PROT_ALL` as defined in '`mach/vm_prot.h`'.

Only data which is cached at the time of this call is affected. When a running thread requires a prohibited access to cached data, the kernel will issue a `memory_object_data_unlock` call specifying the forms of access required.

Once all of the actions requested by this call have been completed, the kernel issues a `memory_object_lock_completed` call on the specified reply port.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. *offset* is an offset within a memory object, in bytes. This must be page aligned. *size* is the amount of cached data (starting at *offset*) to be handled. This must be an integral number of the memory object page size. If *should_clean* is set, modified data should be written back to the memory manager. If *should_flush* is set, the specified cached data should be invalidated, and all uses of that data should be revoked. *lock_value* is a protection value indicating those forms of access that should **not** be permitted to the specified cached data. *reply_to* is a port on which a `memory_object_lock_comleted` call should be issued, or `MACH_PORT_NULL` if no acknowledgement is desired.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

`kern_return_t memory_object_lock_completed` [Function]
(*memory_object_t* `memory_object`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_size_t* `length`)

`kern_return_t seqnos_memory_object_lock_completed` [Function]
(*memory_object_t* `memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_size_t* `length`)

The function `memory_object_lock_completed` indicates that a previous `memory_object_lock_request` has been completed. Note that this call is made on whatever port was specified in the `memory_object_lock_request` call; that port need not be the memory object port itself. No reply is expected after this call.

The argument *memory_object* is the port that represents the memory object data, as supplied to the kernel in a `vm_map` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *offset* is the offset within a memory object to which this call refers. *length* is the length of the data covered by the lock request.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

`kern_return_t memory_object_data_unlock` [Function]
(*memory_object_t* `memory_object`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_size_t* `length`, *vm_prot_t* `desired_access`)

`kern_return_t seqnos_memory_object_data_unlock` [Function]
(*memory_object_t* `memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_size_t* `length`, *vm_prot_t* `desired_access`)

The function `memory_object_data_unlock` is a request that the memory manager permit at least the desired access to the specified data cached by the kernel. A call to `memory_object_lock_request` is expected in response.

The argument *memory_object* is the port that represents the memory object data, as supplied to the kernel in a `vm_map` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *offset* is the offset within a memory object to which this call refers. This will be page aligned. *length* is the number of bytes of data, starting at *offset*, to which this call refers. This will be an integral number of memory object pages. *desired_access* a protection value describing the memory access modes which must be permitted on the specified cached data. One or more of: `VM_PROT_READ`, `VM_PROT_WRITE` or `VM_PROT_EXECUTE`.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

6.6 Memory Object Attributes

`kern_return_t memory_object_get_attributes` [Function]
(*memory_object_control_t* `memory_control`, *boolean_t* *`object_ready`, *boolean_t* *`may_cache_object`, *memory_object_copy_strategy_t* *`copy_strategy`)

The function `memory_object_get_attribute` retrieves the current attributes associated with the memory object.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. If *object_ready* is set, the kernel may issue new data and unlock requests on the associated memory object. If *may_cache_object* is set, the kernel may keep data associated with this memory object, even after virtual memory references to it are gone. *copy_strategy* tells how the kernel should copy regions of the associated memory object.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

`kern_return_t memory_object_change_attributes` [Function]
(*memory_object_control_t* `memory_control`, *boolean_t* `may_cache_object`, *memory_object_copy_strategy_t* `copy_strategy`, *mach_port_t* `reply_to`)

The function `memory_object_change_attribute` sets performance-related attributes for the specified memory object. If the caching attribute is asserted, the kernel is permitted (and encouraged) to maintain cached data for this memory object even after no virtual address space contains this data.

There are three possible caching strategies: `MEMORY_OBJECT_COPY_NONE` which specifies that nothing special should be done when data in the object is copied; `MEMORY_OBJECT_COPY_CALL` which specifies that the memory manager should be notified via a `memory_object_copy` call before any part of the object is copied; and `MEMORY_OBJECT_COPY_DELAY` which guarantees that the memory manager does not externally modify the data so that the kernel can use its normal copy-on-write algorithms. `MEMORY_OBJECT_COPY_DELAY` is the strategy most commonly used.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. If *may_cache_object* is set, the kernel may keep data associated with this memory object, even after virtual memory references to it are gone. *copy_strategy* tells how the kernel should copy regions of the associated memory object. *reply_to* is a port on which a `memory_object_change_comleted` call will be issued upon completion of the attribute change, or `MACH_PORT_NULL` if no acknowledgement is desired.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

`kern_return_t memory_object_change_completed` [Function]
(*memory_object_t* `memory_object`, *boolean_t* `may_cache_object`, *memory_object_copy_strategy_t* `copy_strategy`)

`kern_return_t seqnos_memory_object_change_completed` [Function]
(*memory_object_t* `memory_object`, *mach_port_seqno_t* `seqno`, *boolean_t* `may_cache_object`, *memory_object_copy_strategy_t* `copy_strategy`)

The function `memory_object_change_completed` indicates the completion of an attribute change call.

The following interface is obsoleted by `memory_object_ready` and `memory_object_change_attributes`. If the old form `memory_object_set_attributes` is used to make a memory object ready, the kernel will write back data using the old `memory_object_data_write` interface rather than `memory_object_data_return`..

`kern_return_t memory_object_set_attributes` [Function]
(*memory_object_control_t* `memory_control`, *boolean* `object_ready`, *boolean_t* `may_cache_object`, *memory_object_copy_strategy_t* `copy_strategy`)

The function `memory_object_set_attribute` controls how the the memory object. The kernel will only make data or unlock requests when the ready attribute is asserted. If the caching attribute is asserted, the kernel is permitted (and encouraged) to maintain cached data for this memory object even after no virtual address space contains this data.

There are three possible caching strategies: `MEMORY_OBJECT_COPY_NONE` which specifies that nothing special should be done when data in the object is copied; `MEMORY_OBJECT_COPY_CALL` which specifies that the memory manager should be notified via a `memory_object_copy` call before any part of the object is copied; and `MEMORY_OBJECT_COPY_DELAY` which guarantees that the memory manager does not externally modify the data so that the kernel can use its normal copy-on-write algorithms. `MEMORY_OBJECT_COPY_DELAY` is the strategy most commonly used.

The argument *memory_control* is the port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued. If *object_ready* is set, the kernel may issue new data and unlock requests on the associated memory object. If *may_cache_object* is set, the kernel may keep data associated with this memory object, even after virtual memory references to it are gone. *copy_strategy* tells how the kernel should copy regions of the associated memory object.

This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

6.7 Default Memory Manager

`kern_return_t vm_set_default_memory_manager` (*host_t* `host`, *mach_port_t* `*default_manager`) [Function]

The function `vm_set_default_memory_manager` sets the kernel's default memory manager. It sets the port to which newly-created temporary memory objects are delivered by `memory_object_create` to the host. The old memory manager port is returned. If *default_manager* is `MACH_PORT_NULL` then this routine just returns the current default manager port without changing it.

The argument *host* is a task port to the kernel whose default memory manager is to be changed. *default_manager* is an in/out parameter. As input, *default_manager* is the port that the new memory manager is listening on for `memory_object_create` calls. As output, it is the old default memory manager's port.

The function returns `KERN_SUCCESS` if the new memory manager is installed, and `KERN_INVALID_ARGUMENT` if this task does not have the privileges required for this call.

`kern_return_t memory_object_create` [Function]
(*memory_object_t* `old_memory_object`, *memory_object_t* `new_memory_object`, *vm_size_t* `new_object_size`, *memory_object_control_t* `new_control`, *memory_object_name_t* `new_name`, *vm_size_t* `new_page_size`)

`kern_return_t seqnos_memory_object_create` [Function]
(*memory_object_t* `old_memory_object`, *mach_port_seqno_t* `seqno`, *memory_object_t* `new_memory_object`, *vm_size_t* `new_object_size`, *memory_object_control_t* `new_control`, *memory_object_name_t* `new_name`, *vm_size_t* `new_page_size`)

The function `memory_object_create` is a request that the given memory manager accept responsibility for the given memory object created by the kernel. This call will only be made to the system **default memory manager**. The memory object in question initially consists of zero-filled memory; only memory pages that are actually written will ever be provided to `memory_object_data_request` calls, the default memory manager must use `memory_object_data_unavailable` for any pages that have not previously been written.

No reply is expected after this call. Since this call is directed to the default memory manager, the kernel assumes that it will be ready to handle data requests to this object and does not need the confirmation of a `memory_object_set_attributes` call.

The argument *old_memory_object* is a memory object provided by the default memory manager on which the kernel can make `memory_object_create` calls. *new_memory_object* is a new memory object created by the kernel; see synopsis for further description. Note that all port rights (including receive rights) are included for the new memory object. *new_object_size* is the maximum size of the new object. *new_control* is a port, created by the kernel, on which a memory manager may issue cache management requests for the new object. *new_name* a port used by the kernel to refer to the new memory object data in response to `vm_region` calls. *new_page_size* is the page size to be used by this kernel. All data sizes in calls involving this kernel must be an integral multiple of the page size. Note that different kernels, indicated by different a `memory_control`, may have different page sizes.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

`kern_return_t memory_object_data_initialize` [Function]
(*memory_object_t* `memory_object`, *memory_object_control_t* `memory_control`, *vm_offset_t* `offset`, *vm_offset_t* `data`, *vm_size_t* `data_count`)

`kern_return_t seqnos_memory_object_data_initialize` *(memory_object_t* `memory_object`*, mach_port_seqno_t* `seqno`*, memory_object_control_t* `memory_control`*, vm_offset_t* `offset`*, vm_offset_t* `data`*, vm_size_t* `data_count`*)* [Function]

The function `memory_object_data_initialize` provides the memory manager with initial data for a kernel-created memory object. If the memory manager already has been supplied data (by a previous `memory_object_data_initialize`, `memory_object_data_write` or `memory_object_data_return`), then this data should be ignored. Otherwise, this call behaves exactly as does `memory_object_data_return` on memory objects created by the kernel via `memory_object_create` and thus will only be made to default memory managers. This call will not be made on objects created via `memory_object_copy`.

The argument *memory_object* the port that represents the memory object data, as supplied by the kernel in a `memory_object_create` call. *memory_control* is the request port to which a response is requested. (In the event that a memory object has been supplied to more than one the kernel that has made the request.) *offset* is the offset within a memory object to which this call refers. This will be page aligned. *data* os the data which has been modified while cached in physical memory. *data_count* is the amount of data to be written, in bytes. This will be an integral number of memory object pages.

The function should return `KERN_SUCCESS`, but since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

7 Threads and Tasks

7.1 Thread Interface

`thread_t` [Data type]

This is a `mach_port_t` and used to hold the port name of a thread port that represents the thread. Manipulations of the thread are implemented as remote procedure calls to the thread port. A thread can get a port to itself with the `mach_thread_self` system call.

7.1.1 Thread Creation

`kern_return_t thread_create` (*task_t* `parent_task`, *thread_t* `*child_thread`) [Function]

The function `thread_create` creates a new thread within the task specified by *parent_task*. The new thread has no processor state, and has a suspend count of 1. To get a new thread to run, first `thread_create` is called to get the new thread's identifier, (*child_thread*). Then `thread_set_state` is called to set a processor state, and finally `thread_resume` is called to get the thread scheduled to execute.

When the thread is created send rights to its thread kernel port are given to it and returned to the caller in *child_thread*. The new thread's exception port is set to `MACH_PORT_NULL`.

The function returns `KERN_SUCCESS` if a new thread has been created, `KERN_INVALID_ARGUMENT` if *parent_task* is not a valid task and `KERN_RESOURCE_SHORTAGE` if some critical kernel resource is not available.

7.1.2 Thread Termination

`kern_return_t thread_terminate` (*thread_t* `target_thread`) [Function]

The function `thread_terminate` destroys the thread specified by *target_thread*.

The function returns `KERN_SUCCESS` if the thread has been killed and `KERN_INVALID_ARGUMENT` if *target_thread* is not a thread.

7.1.3 Thread Information

`thread_t mach_thread_self ()` [Function]

The `mach_thread_self` system call returns the calling thread's thread port.

`mach_thread_self` has an effect equivalent to receiving a send right for the thread port. `mach_thread_self` returns the name of the send right. In particular, successive calls will increase the calling task's user-reference count for the send right.

As a special exception, the kernel will overrun the user reference count of the thread name port, so that this function can not fail for that reason. Because of this, the user should not deallocate the port right if an overrun might have happened. Otherwise the reference count could drop to zero and the send right be destroyed while the user still expects to be able to use it. As the kernel does not make use of the number of extant send rights anyway, this is safe to do (the thread port itself is not destroyed, even when there are no send rights anymore).

The function returns `MACH_PORT_NULL` if a resource shortage prevented the reception of the send right or if the thread port is currently null and `MACH_PORT_DEAD` if the thread port is currently dead.

`kern_return_t` **thread_info** (*thread_t* **target_thread**, *int* **flavor**, *thread_info_t* **thread_info**, *mach_msg_type_number_t* ***thread_infoCnt**) [Function]

The function `thread_info` returns the selected information array for a thread, as specified by *flavor*.

thread_info is an array of integers that is supplied by the caller and returned filled with specified information. *thread_infoCnt* is supplied as the maximum number of integers in *thread_info*. On return, it contains the actual number of integers in *thread_info*. The maximum number of integers returned by any flavor is `THREAD_INFO_MAX`.

The type of information returned is defined by *flavor*, which can be one of the following:

`THREAD_BASIC_INFO`
: The function returns basic information about the thread, as defined by `thread_basic_info_t`. This includes the user and system time, the run state, and scheduling priority. The number of integers returned is `THREAD_BASIC_INFO_COUNT`.

`THREAD_SCHED_INFO`
: The function returns information about the schduling policy for the thread as defined by `thread_sched_info_t`. The number of integers returned is `THREAD_SCHED_INFO_COUNT`.

The function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *target_thread* is not a thread or *flavor* is not recognized. The function returns `MIG_ARRAY_TOO_LARGE` if the returned info array is too large for *thread_info*. In this case, *thread_info* is filled as much as possible and *thread_infoCnt* is set to the number of elements that would have been returned if there were enough room.

`struct` **thread_basic_info** [Data type]

This structure is returned in *thread_info* by the `thread_info` function and provides basic information about the thread. You can cast a variable of type `thread_info_t` to a pointer of this type if you provided it as the *thread_info* parameter for the `THREAD_BASIC_INFO` flavor of `thread_info`. It has the following members:

`time_value_t user_time`
: user run time

`time_value_t system_time`
: system run time

`int cpu_usage`
: Scaled cpu usage percentage. The scale factor is `TH_USAGE_SCALE`.

`int base_priority`
: The base scheduling priority of the thread.

`int cur_priority`
: The current scheduling priority of the thread.

`integer_t run_state`
: The run state of the thread. The possible vlues of this field are:

  `TH_STATE_RUNNING`
  : The thread is running normally.

  `TH_STATE_STOPPED`
  : The thread is suspended.

  `TH_STATE_WAITING`
  : The thread is waiting normally.

  `TH_STATE_UNINTERRUPTIBLE`
  : The thread is in an uninterruptible wait.

  `TH_STATE_HALTED`
  : The thread is halted at a clean point.

`flags`
: Various flags. The possible values of this field are:

  `TH_FLAGS_SWAPPED`
  : The thread is swapped out.

  `TH_FLAGS_IDLE`
  : The thread is an idle thread.

`int suspend_count`
: The suspend count for the thread.

`int sleep_time`
: The number of seconds that the thread has been sleeping.

`time_value_t creation_time`
: The time stamp of creation.

`thread_basic_info_t` [Data type]

This is a pointer to a `struct thread_basic_info`.

`struct thread_sched_info` [Data type]

This structure is returned in *thread_info* by the `thread_info` function and provides schedule information about the thread. You can cast a variable of type `thread_info_t` to a pointer of this type if you provided it as the *thread_info* parameter for the `THREAD_SCHED_INFO` flavor of `thread_info`. It has the following members:

`int policy`
: The scheduling policy of the thread, Section 7.1.6.3 [Scheduling Policy], page 71.

`integer_t data`
: Policy-dependent scheduling information, Section 7.1.6.3 [Scheduling Policy], page 71.

`int base_priority`
: The base scheduling priority of the thread.

`int max_priority`
: The maximum scheduling priority of the thread.

`int cur_priority`
: The current scheduling priority of the thread.

`int depressed`
: `TRUE` if the thread is depressed.

`int depress_priority`
: The priority the thread was depressed from.

`thread_sched_info_t` [Data type]

This is a pointer to a `struct thread_sched_info`.

7.1.4 Thread Settings

`kern_return_t thread_wire` (*host_priv_t* `host_priv`, *thread_t* `thread`, *boolean_t* `wired`) [Function]

The function `thread_wire` controls the VM privilege level of the thread *thread*. A VM-privileged thread never waits inside the kernel for memory allocation from the kernel's free list of pages or for allocation of a kernel stack.

Threads that are part of the default pageout path should be VM-privileged, to prevent system deadlocks. Threads that are not part of the default pageout path should not be VM-privileged, to prevent the kernel's free list of pages from being exhausted.

The functions returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_ARGUMENT` if *host_priv* or *thread* was invalid.

The `thread_wire` call is actually an RPC to *host_priv*, normally a send right for a privileged host port, but potentially any send right. In addition to the normal diagnostic return codes from the call's server (normally the kernel), the call may return `mach_msg` return codes.

7.1.5 Thread Execution

`kern_return_t thread_suspend` (*thread_t* `target_thread`) [Function]

Increments the thread's suspend count and prevents the thread from executing any more user level instructions. In this context a user level instruction is either a machine instruction executed in user mode or a system trap instruction including page faults. Thus if a thread is currently executing within a system trap the kernel code may continue to execute until it reaches the system return code or it may supend within the kernel code. In either case, when the thread is resumed the system trap will return. This could cause unpredictible results if the user did a suspend and then altered the user state of the thread in order to change its direction upon a resume. The call `thread_abort` is provided to allow the user to abort any system call that is in progress in a predictable way.

The suspend count may become greater than one with the effect that it will take more than one resume call to restart the thread.

The function returns `KERN_SUCCESS` if the thread has been suspended and `KERN_INVALID_ARGUMENT` if *target_thread* is not a thread.

`kern_return_t thread_resume` (*thread_t* `target_thread`) [Function]

Decrements the threads's suspend count. If the count becomes zero the thread is resumed. If it is still positive, the thread is left suspended. The suspend count may not become negative.

The function returns `KERN_SUCCESS` if the thread has been resumed, `KERN_FAILURE` if the suspend count is already zero and `KERN_INVALID_ARGUMENT` if *target_thread* is not a thread.

`kern_return_t thread_abort` (*thread_t* `target_thread`) [Function]

The function `thread_abort` aborts the kernel primitives: `mach_msg`, `msg_send`, `msg_receive` and `msg_rpc` and page-faults, making the call return a code indicating that it was interrupted. The call is interrupted whether or not the thread (or task containing it) is currently suspended. If it is supsended, the thread receives the interupt when it is resumed.

A thread will retry an aborted page-fault if its state is not modified before it is resumed. `msg_send` returns `SEND_INTERRUPTED`; `msg_receive` returns `RCV_INTERRUPTED`; `msg_rpc` returns either `SEND_INTERRUPTED` or `RCV_INTERRUPTED`, depending on which half of the RPC was interrupted.

The main reason for this primitive is to allow one thread to cleanly stop another thread in a manner that will allow the future execution of the target thread to be controlled in a predictable way. `thread_suspend` keeps the target thread from executing any further instructions at the user level, including the return from a system call. `thread_get_state`/`thread_set_state` allows the examination or modification of the user state of a target thread. However, if a suspended thread was executing within a system call, it also has associated with it a kernel state. This kernel state can not be modified by `thread_set_state` with the result that when the thread is resumed the system call may return changing the user state and possibly user memory. `thread_abort` aborts the kernel call from the target thread's point of view by resetting the kernel state so that the thread will resume execution at the system call return with the return code value set to one of the interrupted codes. The system call itself will either be entirely completed or entirely aborted, depending on the precise moment at which the abort was received. Thus if the thread's user state has been changed by `thread_set_state`, it will not be modified by any unexpected system call side effects.

For example to simulate a Unix signal, the following sequence of calls may be used:

1. `thread_suspend`: Stops the thread.
2. `thread_abort`: Interrupts any system call in progress, setting the return value to 'interrupted'. Since the thread is stopped, it will not return to user code.
3. `thread_set_state`: Alters thread's state to simulate a procedure call to the signal handler
4. `thread_resume`: Resumes execution at the signal handler. If the thread's stack has been correctly set up, the thread may return to the interrupted system call. (Of course, the code to push an extra stack frame and change the registers is VERY machine-dependent.)

Calling `thread_abort` on a non-suspended thread is pretty risky, since it is very difficult to know exactly what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

The function returns `KERN_SUCCESS` if the thread received an interrupt and `KERN_INVALID_ARGUMENT` if *target_thread* is not a thread.

`kern_return_t thread_get_state` (*thread_t* `target_thread`, *int* `flavor`, *thread_state_t* `old_state`, *mach_msg_type_number_t* `*old_stateCnt`) [Function]

The function `thread_get_state` returns the execution state (e.g. the machine registers) of *target_thread* as specified by *flavor*. The *old_state* is an array of integers that is provided by the caller and returned filled with the specified information. *old_stateCnt* is input set to the maximum number of integers in *old_state* and returned equal to the actual number of integers in *old_state*.

target_thread may not be `mach_thread_self()`.

The definition of the state structures can be found in '`machine/thread_status.h`'.

The function returns `KERN_SUCCESS` if the state has been returned, `KERN_INVALID_ARGUMENT` if *target_thread* is not a thread or is `mach_thread_self` or *flavor* is unrecogized for this machine. The function returns `MIG_ARRAY_TOO_LARGE` if the returned state is too large for *old_state*. In this case, *old_state* is filled as much as possible and *old_stateCnt* is set to the number of elements that would have been returned if there were enough room.

`kern_return_t thread_set_state` (*thread_t* `target_thread`, *int* `flavor`, *thread_state_t* `new_state`, *mach_msg_type_number_t* `new_state_count`) [Function]

The function `thread_set_state` sets the execution state (e.g. the machine registers) of *target_thread* as specified by *flavor*. The *new_state* is an array of integers. *new_state_count* is the number of elements in *new_state*. The entire set of registers is reset. This will do unpredictable things if *target_thread* is not suspended.

target_thread may not be `mach_thread_self`.

The definition of the state structures can be found in '`machine/thread_status.h`'.

The function returns `KERN_SUCCESS` if the state has been set and `KERN_INVALID_ARGUMENT` if *target_thread* is not a thread or is `mach_thread_self` or *flavor* is unrecogized for this machine.

7.1.6 Scheduling

7.1.6.1 Thread Priority

Threads have three priorities associated with them by the system, a priority, a maximum priority, and a scheduled priority. The scheduled priority is used to make scheduling decisions about the thread. It is determined from the priority by the policy (for timesharing, this means adding an increment derived from cpu usage). The priority can be set under user control, but may never exceed the maximum priority. Changing the maximum priority requires presentation of the control port for the thread's processor set; since the control port for the default processor set is privileged, users cannot raise their maximum priority to unfairly compete with other users on that set. Newly created threads obtain their priority from their task and their max priority from the thread.

`kern_return_t thread_priority` (*thread_t* `thread`, *int* `prority`, *boolean_t* `set_max`) [Function]

The function `thread_priority` changes the priority and optionally the maximum priority of *thread*. Priorities range from 0 to 31, where lower numbers denote higher priorities. If the new priority is higher than the priority of the current thread, preemption may occur as a result of this call. The maximum priority of the thread is also set if *set_max* is `TRUE`. This call will fail if *priority* is greater than the current maximum priority of the thread. As a result, this call can only lower the value of a thread's maximum priority.

The functions returns `KERN_SUCCESS` if the operation completed successfully, `KERN_INVALID_ARGUMENT` if *thread* is not a thread or *priority* is out of range (not in 0..31), and `KERN_FAILURE` if the requested operation would violate the thread's maximum priority (thread_priority).

`kern_return_t thread_max_priority` (*thread_t* `thread`, *processor_set_t* `processor_set`, *int* `priority`) [Function]

The function `thread_max_priority` changes the maximum priority of the thread. Because it requires presentation of the corresponding processor set port, this call can reset the maximum priority to any legal value.

The functions returns `KERN_SUCCESS` if the operation completed successfully, `KERN_INVALID_ARGUMENT` if *thread* is not a thread or *processor_set* is not a control port for a processor set or *priority* is out of range (not in 0..31), and `KERN_FAILURE` if the thread is not assigned to the processor set whose control port was presented.

7.1.6.2 Hand-Off Scheduling

`kern_return_t thread_switch` (*thread_t* `new_thread`, *int* `option`, *int* `time`) [Function]

The function `thread_switch` provides low-level access to the scheduler's context switching code. *new_thread* is a hint that implements hand-off scheduling. The operating system will attempt to switch directly to the new thread (by passing the normal logic that selects the next thread to run) if possible. Since this is a hint, it may be incorrect; it is ignored if it doesn't specify a thread on the same host as the current thread or if that thread can't be switched to (i.e., not runnable or already running on another processor). In this case, the normal logic to select the next thread to run is used; the current thread may continue running if there is no other appropriate thread to run.

Options for *option* are defined in '`mach/thread_switch.h`' and specify the interpretation of *time*. The possible values for *option* are:

`SWITCH_OPTION_NONE`
: No options, the time argument is ignored.

`SWITCH_OPTION_WAIT`
: The thread is blocked for the specified time. This can be aborted by `thread_abort`.

`SWITCH_OPTION_DEPRESS`
: The thread's priority is depressed to the lowest possible value for the specified time. This can be aborted by `thread_depress_abort`. This depression is independent of operations that change the thread's priority (e.g. `thread_priority` will not abort the depression). The minimum time and units of time can be obtained as the `min_timeout` value from `host_info`. The depression is also aborted when the current thread is next run (either via handoff scheduling or because the processor set has nothing better to do).

`thread_switch` is often called when the current thread can proceed no further for some reason; the various options and arguments allow information about this reason to be transmitted to the kernel. The *new_thread* argument (handoff scheduling) is useful when the identity of the thread that must make progress before the current thread runs again is known. The `WAIT` option is used when the amount of time that the current thread must wait before it can do anything useful can be estimated and is fairly long. The `DEPRESS` option is used when the amount of time that must be waited is fairly short, especially when the identity of the thread that is being waited for is not known.

Users should beware of calling `thread_switch` with an invalid hint (e.g. `MACH_PORT_NULL`) and no option. Because the time-sharing scheduler varies the priority of threads based on usage, this may result in a waste of cpu time if the thread that must be run is of lower priority. The use of the `DEPRESS` option in this situation is highly recommended.

`thread_switch` ignores policies. Users relying on the preemption semantics of a fixed time policy should be aware that `thread_switch` ignores these semantics; it will run the specified *new_thread* indepent of its priority and the priority of any other threads that could be run instead.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_ARGUMENT` if *thread* is not a thread or *option* is not a recognized option, and `KERN_FAILURE` if `kern_depress_abort` failed because the thread was not depressed.

`kern_return_t thread_depress_abort` (*thread_t* `thread`) [Function]

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

The function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *thread* is not a valid thread.

`boolean_t swtch` () [Function]

The system trap `swtch` attempts to switch the current thread off the processor. The return value indicates if more than the current thread is running in the processor set. This is useful for lock management routines.

The call returns `FALSE` if the thread is justified in becoming a resource hog by continuing to spin because there's nothing else useful that the processor could do. `TRUE` is returned if the thread should make one more check on the lock and then be a good citizen and really suspend.

`boolean_t swtch_pri` (*int* `priority`) [Function]

The system trap `swtch_pri` attempts to switch the current thread off the processor as `swtch` does, but depressing the priority of the thread to the minimum possible value during the time. *priority* is not used currently.

The return value is as for `swtch`.

7.1.6.3 Scheduling Policy

`kern_return_t thread_policy` (*thread_t* `thread`, *int* `policy`, *int* `data`) [Function]

The function `thread_policy` changes the scheduling policy for *thread* to *policy*.

data is policy-dependent scheduling information. There are currently two supported policies: `POLICY_TIMESHARE` and `POLICY_FIXEDPRI` defined in '`mach/policy.h`'; this file is included by '`mach.h`'. *data* is meaningless for timesharing, but is the quantum to be used (in milliseconds) for the fixed priority policy. To be meaningful, this quantum must be a multiple of the basic system quantum (min_quantum) which can be obtained from `host_info`. The system will always round up to the next multiple of the quantum.

Processor sets may restrict the allowed policies, so this call will fail if the processor set to which *thread* is currently assigned does not permit *policy*.

The function returns `KERN_SUCCESS` if the call succeeded. `KERN_INVALID_ARGUMENT` if *thread* is not a thread or *policy* is not a recognized policy, and `KERN_FAILURE` if the processor set to which *thread* is currently assigned does not permit *policy*.

7.1.7 Thread Special Ports

`kern_return_t thread_get_special_port` (*thread_t* `thread`, *int* `which_port`, *mach_port_t* `*special_port`) [Function]

The function `thread_get_special_port` returns send rights to one of a set of special ports for the thread specified by *thread*.

The possible values for *which_port* are `THREAD_KERNEL_PORT` and `THREAD_EXCEPTION_PORT`. A thread also has access to its task's special ports.

The function returns `KERN_SUCCESS` if the port was returned and `KERN_INVALID_ARGUMENT` if *thread* is not a thread or *which_port* is an invalid port selector.

`kern_return_t thread_get_kernel_port` (*thread_t* `thread`, *mach_port_t* `*kernel_port`) [Function]

The function `thread_get_kernel_port` is equivalent to the function `thread_get_special_port` with the *which_port* argument set to `THREAD_KERNEL_PORT`.

`kern_return_t thread_get_exception_port` (*thread_t* `thread`, *mach_port_t* `*exception_port`) [Function]

The function `thread_get_exception_port` is equivalent to the function `thread_get_special_port` with the *which_port* argument set to `THREAD_EXCEPTION_PORT`.

`kern_return_t thread_set_special_port` (*thread_t* `thread`, *int* `which_port`, *mach_port_t* `special_port`) [Function]

The function `thread_set_special_port` sets one of a set of special ports for the thread specified by *thread*.

The possible values for *which_port* are `THREAD_KERNEL_PORT` and `THREAD_EXCEPTION_PORT`. A thread also has access to its task's special ports.

The function returns `KERN_SUCCESS` if the port was set and `KERN_INVALID_ARGUMENT` if *thread* is not a thread or *which_port* is an invalid port selector.

`kern_return_t thread_set_kernel_port` (*thread_t* **`thread`**, *mach_port_t* **`kernel_port`**) [Function]

The function `thread_set_kernel_port` is equivalent to the function `thread_set_special_port` with the *which_port* argument set to `THREAD_KERNEL_PORT`.

`kern_return_t thread_set_exception_port` (*thread_t* **`thread`**, *mach_port_t* **`exception_port`**) [Function]

The function `thread_set_exception_port` is equivalent to the function `thread_set_special_port` with the *which_port* argument set to `THREAD_EXCEPTION_PORT`.

7.1.8 Exceptions

`kern_return_t catch_exception_raise` (*mach_port_t* **`exception_port`**, *thread_t* **`thread`**, *task_t* **`task`**, *int* **`exception`**, *int* **`code`**, *int* **`subcode`**) [Function]

XXX Fixme

`kern_return_t exception_raise` (*mach_port_t* **`exception_port`**, *mach_port_t* **`thread`**, *mach_port_t* **`task`**, *integer_t* **`exception`**, *integer_t* **`code`**, *integer_t* **`subcode`**) [Function]

XXX Fixme

`kern_return_t evc_wait` (*unsigned int* **`event`**) [Function]

The system trap `evc_wait` makes the calling thread wait for the event specified by *event*.

The call returns `KERN_SUCCESS` if the event has occured, `KERN_NO_SPACE` if another thread is waiting for the same event and `KERN_INVALID_ARGUMENT` if the event object is invalid.

7.2 Task Interface

`task_t` [Data type]

This is a `mach_port_t` and used to hold the port name of a task port that represents the thread. Manipulations of the task are implemented as remote procedure calls to the task port. A task can get a port to itself with the `mach_task_self` system call.

The task port name is also used to identify the task's IPC space (see Section 4.3 [Port Manipulation Interface], page 29) and the task's virtual memory map (see Chapter 5 [Virtual Memory Interface], page 41).

7.2.1 Task Creation

`kern_return_t task_create (task_t parent_task, boolean_t inherit_memory, task_t *child_task)` [Function]

The function `task_create` creates a new task from *parent_task*; the resulting task (*child_task*) acquires shared or copied parts of the parent's address space (see `vm_inherit`). The child task initially contains no threads.

If *inherit_memory* is set, the child task's address space is built from the parent task according to its memory inheritance values; otherwise, the child task is given an empty address space.

The child task gets the three special ports created or copied for it at task creation. The `TASK_KERNEL_PORT` is created and send rights for it are given to the child and returned to the caller. The `TASK_BOOTSTRAP_PORT` and the `TASK_EXCEPTION_PORT` are inherited from the parent task. The new task can get send rights to these ports with the call `task_get_special_port`.

The function returns `KERN_SUCCESS` if a new task has been created, `KERN_INVALID_ARGUMENT` if *parent_task* is not a valid task port and `KERN_RESOURCE_SHORTAGE` if some critical kernel resource is unavailable.

7.2.2 Task Termination

`kern_return_t task_terminate (task_t target_task)` [Function]

The function `task_terminate` destroys the task specified by *target_task* and all its threads. All resources that are used only by this task are freed. Any port to which this task has receive and ownership rights is destroyed.

The function returns `KERN_SUCCESS` if the task has been killed, `KERN_INVALID_ARGUMENT` if *target_task* is not a task.

7.2.3 Task Information

`task_t mach_task_self ()` [Function]

The `mach_task_self` system call returns the calling thread's task port.

`mach_task_self` has an effect equivalent to receiving a send right for the task port. `mach_task_self` returns the name of the send right. In particular, successive calls will increase the calling task's user-reference count for the send right.

As a special exception, the kernel will overrun the user reference count of the task name port, so that this function can not fail for that reason. Because of this, the user should not deallocate the port right if an overrun might have happened. Otherwise the reference count could drop to zero and the send right be destroyed while the user still expects to be able to use it. As the kernel does not make use of the number of extant send rights anyway, this is safe to do (the task port itself is not destroyed, even when there are no send rights anymore).

The funcion returns `MACH_PORT_NULL` if a resource shortage prevented the reception of the send right, `MACH_PORT_NULL` if the task port is currently null, `MACH_PORT_DEAD` if the task port is currently dead.

`kern_return_t task_threads` (*task_t* `target_task`, *thread_array_t* `*thread_list`, *mach_msg_type_number_t* `*thread_count`) [Function]

The function `task_threads` gets send rights to the kernel port for each thread contained in *target_task*. *thread_list* is an array that is created as a result of this call. The caller may wish to `vm_deallocate` this array when the data is no longer needed.

The function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *target_task* is not a task.

`kern_return_t task_info` (*task_t* `target_task`, *int* `flavor`, *task_info_t* `task_info`, *mach_msg_type_number_t* `*task_info_count`) [Function]

The function `task_info` returns the selected information array for a task, as specified by *flavor*. *task_info* is an array of integers that is supplied by the caller, and filled with specified information. *task_info_count* is supplied as the maximum number of integers in *task_info*. On return, it contains the actual number of integers in *task_info*. The maximum number of integers returned by any flavor is `TASK_INFO_MAX`.

The type of information returned is defined by *flavor*, which can be one of the following:

`TASK_BASIC_INFO`
: The function returns basic information about the task, as defined by `task_basic_info_t`. This includes the user and system time and memory consumption. The number of integers returned is `TASK_BASIC_INFO_COUNT`.

`TASK_EVENTS_INFO`
: The function returns information about events for the task as defined by `thread_sched_info_t`. This includes statistics about virtual memory and IPC events like pageouts, pageins and messages sent and received. The number of integers returned is `TASK_EVENTS_INFO_COUNT`.

`TASK_THREAD_TIMES_INFO`
: The function returns information about the total time for live threads as defined by `task_thread_times_info_t`. The number of integers returned is `TASK_THREAD_TIMES_INFO_COUNT`.

The function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *target_task* is not a thread or *flavor* is not recognized. The function returns `MIG_ARRAY_TOO_LARGE` if the returned info array is too large for *task_info*. In this case, *task_info* is filled as much as possible and *task_infoCnt* is set to the number of elements that would have been returned if there were enough room.

`struct task_basic_info` [Data type]

This structure is returned in *task_info* by the `task_info` function and provides basic information about the task. You can cast a variable of type `task_info_t` to a pointer of this type if you provided it as the *task_info* parameter for the `TASK_BASIC_INFO` flavor of `task_info`. It has the following members:

`integer_t suspend_count`
: suspend count for task

`integer_t base_priority`
: base scheduling priority

`vm_size_t virtual_size`
: number of virtual pages

`vm_size_t resident_size`
: number of resident pages

`time_value_t user_time`
: total user run time for terminated threads

`time_value_t system_time`
: total system run time for terminated threads

`time_value_t creation_time`
: creation time stamp

`task_basic_info_t` [Data type]

This is a pointer to a `struct task_basic_info`.

`struct task_events_info` [Data type]

This structure is returned in *task_info* by the `task_info` function and provides event statistics for the task. You can cast a variable of type `task_info_t` to a pointer of this type if you provided it as the *task_info* parameter for the `TASK_EVENTS_INFO` flavor of `task_info`. It has the following members:

`natural_t faults`
: number of page faults

`natural_t zero_fills`
: number of zero fill pages

`natural_t reactivations`
: number of reactivated pages

`natural_t pageins`
: number of actual pageins

`natural_t cow_faults`
: number of copy-on-write faults

`natural_t messages_sent`
: number of messages sent

`natural_t messages_received`
: number of messages received

`task_events_info_t` [Data type]

This is a pointer to a `struct task_events_info`.

`struct task_thread_times_info` [Data type]

This structure is returned in *task_info* by the `task_info` function and provides event statistics for the task. You can cast a variable of type `task_info_t` to a pointer of this type if you provided it as the *task_info* parameter for the `TASK_THREAD_TIMES_INFO` flavor of `task_info`. It has the following members:

`time_value_t user_time`
: total user run time for live threads

`time_value_t system_time`
: total system run time for live threads

`task_thread_times_info_t` [Data type]

This is a pointer to a `struct task_thread_times_info`.

7.2.4 Task Execution

`kern_return_t task_suspend` (*task_t* `target_task`) [Function]

The function `task_suspend` increments the task's suspend count and stops all threads in the task. As long as the suspend count is positive newly created threads will not run. This call does not return until all threads are suspended.

The count may become greater than one, with the effect that it will take more than one resume call to restart the task.

The function returns `KERN_SUCCESS` if the task has been suspended and `KERN_INVALID_ARGUMENT` if *target_task* is not a task.

`kern_return_t task_resume` (*task_t* `target_task`) [Function]

The function `task_resume` decrements the task's suspend count. If it becomes zero, all threads with zero suspend counts in the task are resumed. The count may not become negative.

The function returns `KERN_SUCCESS` if the task has been resumed, `KERN_FAILURE` if the suspend count is already at zero and `KERN_INVALID_ARGUMENT` if *target_task* is not a task.

`kern_return_t task_priority` (*task_t* `task`, *int* `priority`, *boolean_t* `change_threads`) [Function]

The priority of a task is used only for creation of new threads; a new thread's priority is set to the enclosing task's priority. `task_priority` changes this task priority. It also sets the priorities of all threads in the task to this new priority if *change_threads* is `TRUE`. Existing threads are not affected otherwise. If this priority change violates the maximum priority of some threads, as many threads as possible will be changed and an error code will be returned.

The function returns `KERN_SUCCESS` if the call succeeded, `KERN_INVALID_ARGUMENT` if *task* is not a task, or *priority* is not a valid priority and `KERN_FAILURE` if *change_threads* was `TRUE` and the attempt to change the priority of at least one existing thread failed because the new priority would have exceeded that thread's maximum priority.

`kern_return_t task_ras_control` (*task_t* `target_task`, *vm_address_t* `start_pc`, *vm_address_t* `end_pc`, *int* `flavor`) [Function]

The function `task_ras_control` manipulates a task's set of restartable atomic sequences. If a sequence is installed, and any thread in the task is preempted within the range [*start_pc*,*end_pc*], then the thread is resumed at *start_pc*. This enables applications to build atomic sequences which, when executed to completion, will have executed atomically. Restartable atomic sequences are intended to be used on systems that do not have hardware support for low-overhead atomic primitives.

As a thread can be rolled-back, the code in the sequence should have no side effects other than a final store at *end_pc*. The kernel does not guarantee that the sequence is restartable. It assumes the application knows what it's doing.

A task may have a finite number of atomic sequences that is defined at compile time.

The flavor specifices the particular operation that should be applied to this restartable atomic sequence. Possible values for flavor can be:

`TASK_RAS_CONTROL_PURGE_ALL`
: Remove all registered sequences for this task.

`TASK_RAS_CONTROL_PURGE_ONE`
: Remove the named registered sequence for this task.

`TASK_RAS_CONTROL_PURGE_ALL_AND_INSTALL_ONE`
: Atomically remove all registered sequences and install the named sequence.

`TASK_RAS_CONTROL_INSTALL_ONE`
: Install this sequence.

The function returns `KERN_SUCCESS` if the operation has been performed, `KERN_INVALID_ADDRESS` if the *start_pc* or *end_pc* values are not a valid address for the requested operation (for example, it is invalid to purge a sequence that has not been registered), `KERN_RESOURCE_SHORTAGE` if an attempt was made to install more restartable atomic sequences for a task than can be supported by the kernel, `KERN_INVALID_VALUE` if a bad flavor was specified, `KERN_INVALID_ARGUMENT` if *target_task* is not a task and `KERN_FAILURE` if the call is not not supported on this configuration.

7.2.5 Task Special Ports

`kern_return_t task_get_special_port` (*task_t* `task`, *int* `which_port`, *mach_port_t* `*special_port`) [Function]

The function `task_get_special_port` returns send rights to one of a set of special ports for the task specified by *task*.

The special ports associated with a task are the kernel port (`TASK_KERNEL_PORT`), the bootstrap port (`TASK_BOOTSTRAP_PORT`) and the exception port (`TASK_EXCEPTION_PORT`). The bootstrap port is a port to which a task may send a message requesting other system service ports. This port is not used by the kernel. The task's exception port is the port to which messages are sent by the kernel when an exception occurs and the thread causing the exception has no exception port of its own.

The following macros to call `task_get_special_port` for a specific port are defined in `mach/task_special_ports.h`: `task_get_exception_port` and `task_get_bootstrap_port`.

The function returns `KERN_SUCCESS` if the port was returned and `KERN_INVALID_ARGUMENT` if *task* is not a task or *which_port* is an invalid port selector.

kern_return_t task_get_kernel_port (*task_t* task, *mach_port_t* *kernel_port) [Function]

The function task_get_kernel_port is equivalent to the function task_get_special_port with the *which_port* argument set to TASK_KERNEL_PORT.

kern_return_t task_get_exception_port (*task_t* task, *mach_port_t* *exception_port) [Function]

The function task_get_exception_port is equivalent to the function task_get_special_port with the *which_port* argument set to TASK_EXCEPTION_PORT.

kern_return_t task_get_bootstrap_port (*task_t* task, *mach_port_t* *bootstrap_port) [Function]

The function task_get_bootstrap_port is equivalent to the function task_get_special_port with the *which_port* argument set to TASK_BOOTSTRAP_PORT.

kern_return_t task_set_special_port (*task_t* task, *int* which_port, *mach_port_t* special_port) [Function]

The function thread_set_special_port sets one of a set of special ports for the task specified by *task*.

The special ports associated with a task are the kernel port (TASK_KERNEL_PORT), the bootstrap port (TASK_BOOTSTRAP_PORT) and the exception port (TASK_EXCEPTION_PORT). The bootstrap port is a port to which a thread may send a message requesting other system service ports. This port is not used by the kernel. The task's exception port is the port to which messages are sent by the kernel when an exception occurs and the thread causing the exception has no exception port of its own.

The function returns KERN_SUCCESS if the port was set and KERN_INVALID_ARGUMENT if *task* is not a task or *which_port* is an invalid port selector.

kern_return_t task_set_kernel_port (*task_t* task, *mach_port_t* kernel_port) [Function]

The function task_set_kernel_port is equivalent to the function task_set_special_port with the *which_port* argument set to TASK_KERNEL_PORT.

kern_return_t task_set_exception_port (*task_t* task, *mach_port_t* exception_port) [Function]

The function task_set_exception_port is equivalent to the function task_set_special_port with the *which_port* argument set to TASK_EXCEPTION_PORT.

kern_return_t task_set_bootstrap_port (*task_t* task, *mach_port_t* bootstrap_port) [Function]

The function task_set_bootstrap_port is equivalent to the function task_set_special_port with the *which_port* argument set to TASK_BOOTSTRAP_PORT.

7.2.6 Syscall Emulation

kern_return_t task_get_emulation_vector (*task_t* task, *int* *vector_start, *emulation_vector_t* *emulation_vector, *mach_msg_type_number_t* *emulation_vector_count) [Function]

The function task_get_emulation_vector gets the user-level handler entry points for all emulated system calls.

kern_return_t **task_set_emulation_vector** (*task_t* **task**, *int* **vector_start**, *emulation_vector_t* **emulation_vector**, *mach_msg_type_number_t* **emulation_vector_count**) [Function]

The function `task_set_emulation_vector` establishes user-level handlers for the specified system calls. Non-emulated system calls are specified with an entry of `EML_ROUTINE_NULL`. System call emulation handlers are inherited by the childs of *task*.

kern_return_t **task_set_emulation** (*task_t* **task**, *vm_address_t* **routine_entry_pt**, *int* **routine_number**) [Function]

The function `task_set_emulation` establishes a user-level handler for the specified system call. System call emulation handlers are inherited by the childs of *task*.

7.3 Profiling

kern_return_t **task_enable_pc_sampling** (*task_t* **task**, *int* ***ticks**, *sampled_pc_flavor_t* **flavor**) [Function]

kern_return_t **thread_enable_pc_sampling** (*thread_t* **thread**, *int* ***ticks**, *sampled_pc_flavor_t* **flavor**) [Function]

The function `task_enable_pc_sampling` enables PC sampling for *task*, the function `thread_enable_pc_sampling` enables PC sampling for *thread*. The kernel's idea of clock granularity is returned in *ticks* in usecs. (this value should not be trusted). The sampling flavor is specified by *flavor*.

The function returns `KERN_SUCCESS` if the operation is completed successfully and `KERN_INVALID_ARGUMENT` if *thread* is not a valid thread.

kern_return_t **task_disable_pc_sampling** (*task_t* **task**, *int* ***sample_count**) [Function]

kern_return_t **thread_disable_pc_sampling** (*thread_t* **thread**, *int* ***sample_count**) [Function]

The function `task_disable_pc_sampling` disables PC sampling for *task*, the function `thread_disable_pc_sampling` disables PC sampling for *thread*. The number of sample elements in the kernel for the thread is returned in *sample_count*.

The function returns `KERN_SUCCESS` if the operation is completed successfully and `KERN_INVALID_ARGUMENT` if *thread* is not a valid thread.

kern_return_t **task_get_sampled_pcs** (*task_t* **task**, *sampled_pc_seqno_t* ***seqno**, *sampled_pc_array_t* **sampled_pcs**, *mach_msg_type_number_t* ***sample_count**) [Function]

kern_return_t **thread_get_sampled_pcs** (*thread_t* **thread**, *sampled_pc_seqno_t* ***seqno**, *sampled_pc_array_t* **sampled_pcs**, *int* ***sample_count**) [Function]

The function `task_get_sampled_pcs` extracts the PC samples for *task*, the function `thread_get_sampled_pcs` extracts the PC samples for *thread*. *seqno* is the sequence number of the sampled PCs. This is useful for determining when a collector thread has missed a sample. The sampled PCs for the thread are returned in *sampled_pcs*. *sample_count* contains the number of sample elements returned.

The function returns KERN_SUCCESS if the operation is completed successfully, KERN_INVALID_ARGUMENT if *thread* is not a valid thread and KERN_FAILURE if *thread* is not sampled.

sampled_pc_t [Data type]

This structure is returned in *sampled_pcs* by the thread_get_sampled_pcs and task_get_sampled_pcs functions and provides pc samples for threads or tasks. It has the following members:

natural_t id
: A thread-specific unique identifier.

vm_offset_t pc
: A pc value.

sampled_pc_flavor_t sampletype
: The type of the sample as per flavor.

sampled_pc_flavor_t [Data type]

This data type specifies a pc sample flavor, either as argument passed in *flavor* to the thread_enable_pc_sample and thread_disable_pc_sample functions, or as member sampletype in the sample_pc_t data type. The flavor is a bitwise-or of the possible flavors defined in 'mach/pc_sample.h':

SAMPLED_PC_PERIODIC
: default

SAMPLED_PC_VM_ZFILL_FAULTS
: zero filled fault

SAMPLED_PC_VM_REACTIVATION_FAULTS
: reactivation fault

SAMPLED_PC_VM_PAGEIN_FAULTS
: pagein fault

SAMPLED_PC_VM_COW_FAULTS
: copy-on-write fault

SAMPLED_PC_VM_FAULTS_ANY
: any fault

SAMPLED_PC_VM_FAULTS
: the bitwise-or of SAMPLED_PC_VM_ZFILL_FAULTS, SAMPLED_PC_VM_REACTIVATION_FAULTS, SAMPLED_PC_VM_PAGEIN_FAULTS and SAMPLED_PC_VM_COW_FAULTS.

8 Host Interface

This section describes the Mach interface to a host executing a Mach kernel. The interface allows to query statistics about a host and control its behaviour.

A host is represented by two ports, a name port *host* used to query information about the host accessible to everyone, and a control port *host_priv* used to manipulate it. For example, you can query the current time using the name port, but to change the time you need to send a message to the host control port.

Everything described in this section is declared in the header file 'mach.h'.

8.1 Host Ports

`host_t` [Data type]

This is a `mach_port_t` and used to hold the port name of a host name port (or short: host port). Any task can get a send right to the name port of the host running the task using the `mach_host_self` system call. The name port can be used query information about the host, for example the current time.

`host_t mach_host_self ()` [Function]

The `mach_host_self` system call returns the calling thread's host name port. It has an effect equivalent to receiving a send right for the host port. `mach_host_self` returns the name of the send right. In particular, successive calls will increase the calling task's user-reference count for the send right.

As a special exception, the kernel will overrun the user reference count of the host name port, so that this function can not fail for that reason. Because of this, the user should not deallocate the port right if an overrun might have happened. Otherwise the reference count could drop to zero and the send right be destroyed while the user still expects to be able to use it. As the kernel does not make use of the number of extant send rights anyway, this is safe to do (the host port itself is never destroyed).

The function returns `MACH_PORT_NULL` if a resource shortage prevented the reception of the send right.

This function is also available in 'mach/mach_traps.h'.

`host_priv_t` [Data type]

This is a `mach_port_t` and used to hold the port name of a privileged host control port. A send right to the host control port is inserted into the first task at bootstrap (see Section 3.2 [Modules], page 11). This is the only way to get access to the host control port in Mach, so the initial task has to preserve the send right carefully, moving a copy of it to other privileged tasks if necessary and denying access to unprivileged tasks.

8.2 Host Information

`kern_return_t host_info` (*host_t* **host**, *int* **flavor**, *host_info_t* **host_info**, *mach_msg_type_number_t* ***host_info_count**) [Function]

The `host_info` function returns various information about *host*. *host_info* is an array of integers that is supplied by the caller. It will be filled with the requested information. *host_info_count* is supplied as the maximum number of integers in *host_info*. On return, it contains the actual number of integers in *host_info*. The maximum number of integers returned by any flavor is `HOST_INFO_MAX`.

The type of information returned is defined by *flavor*, which can be one of the following:

`HOST_BASIC_INFO`
: The function returns basic information about the host, as defined by `host_basic_info_t`. This includes the number of processors, their type, and the amount of memory installed in the system. The number of integers returned is `HOST_BASIC_INFO_COUNT`. For how to get more information about the processor, see Section 9.2 [Processor Interface], page 92.

`HOST_PROCESSOR_SLOTS`
: The function returns the numbers of the slots with active processors in them. The number of integers returned can be up to `max_cpus`, as returned by the `HOST_BASIC_INFO` flavor of `host_info`.

`HOST_SCHED_INFO`
: The function returns information of interest to schedulers as defined by `host_sched_info_t`. The number of integers returned is `HOST_SCHED_INFO_COUNT`.

The function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *host* is not a host or *flavor* is not recognized. The function returns `MIG_ARRAY_TOO_LARGE` if the returned info array is too large for *host_info*. In this case, *host_info* is filled as much as possible and *host_info_count* is set to the number of elements that would be returned if there were enough room.

`struct host_basic_info` [Data type]

A pointer to this structure is returned in *host_info* by the `host_info` function and provides basic information about the host. You can cast a variable of type `host_info_t` to a pointer of this type if you provided it as the *host_info* parameter for the `HOST_BASIC_INFO` flavor of `host_info`. It has the following members:

`int max_cpus`
: The maximum number of possible processors for which the kernel is configured.

`int avail_cpus`
: The number of cpus currently available.

`vm_size_t memory_size`
: The size of physical memory in bytes.

`cpu_type_t cpu_type`
: The type of the master processor.

`cpu_subtype_t cpu_subtype`
: The subtype of the master processor.

The type and subtype of the individual processors are also available by `processor_info`, see Section 9.2 [Processor Interface], page 92.

`host_basic_info_t` [Data type]

This is a pointer to a `struct host_basic_info`.

`struct host_sched_info` [Data type]

A pointer to this structure is returned in *host_info* by the `host_info` function and provides information of interest to schedulers. You can cast a variable of type `host_info_t` to a pointer of this type if you provided it as the *host_info* parameter for the `HOST_SCHED_INFO` flavor of `host_info`. It has the following members:

`int min_timeout`
: The minimum timeout and unit of time in milliseconds.

`int min_quantum`
: The minimum quantum and unit of quantum in milliseconds.

`host_sched_info_t` [Data type]

This is a pointer to a `struct host_sched_info`.

`kern_return_t host_kernel_version (`*host_t* ***host***, *kernel_version_t* ****version***`)` [Function]

The `host_kernel_version` function returns the version string compiled into the kernel executing on *host* at the time it was built in the character string *version*. This string describes the version of the kernel. The constant `KERNEL_VERSION_MAX` should be used to dimension storage for the returned string if the `kernel_version_t` declaration is not used.

If the version string compiled into the kernel is longer than `KERNEL_VERSION_MAX`, the result is truncated and not necessarily null-terminated.

If *host* is not a valid send right to a host port, the function returns `KERN_INVALID_ARGUMENT`. If *version* points to inaccessible memory, it returns `KERN_INVALID_ADDRESS`, and `KERN_SUCCESS` otherwise.

`kern_return_t host_get_boot_info (`*host_priv_t* ***host_priv***, *kernel_boot_info_t* ***boot_info***`)` [Function]

The `host_get_boot_info` function returns the boot-time information string supplied by the operator to the kernel executing on *host_priv* in the character string *boot_info*. The constant `KERNEL_BOOT_INFO_MAX` should be used to dimension storage for the returned string if the `kernel_boot_info_t` declaration is not used.

If the boot-time information string supplied by the operator is longer than `KERNEL_BOOT_INFO_MAX`, the result is truncated and not necessarily null-terminated.

8.3 Host Time

`time_value_t` [Data type]

This is the representation of a time in Mach. It is a `struct time_value` and consists of the following members:

`integer_t seconds`
: The number of seconds.

`integer_t microseconds`
: The number of microseconds.

The number of microseconds should always be smaller than `TIME_MICROS_MAX` (100000). A time with this property is *normalized*. Normalized time values can be manipulated with the following macros:

`time_value_add_usec` (*time_value_t* `*val`, *integer_t* `*micros`) [Macro]
: Add *micros* microseconds to *val*. If *val* is normalized and *micros* smaller than `TIME_MICROS_MAX`, *val* will be normalized afterwards.

`time_value_add` (*time_value_t* `*result`, *time_value_t* `*addend`) [Macro]
: Add the values in *addend* to *result*. If both are normalized, *result* will be normalized afterwards.

A variable of type `time_value_t` can either represent a duration or a fixed point in time. In the latter case, it shall be interpreted as the number of seconds and microseconds after the epoch 1. Jan 1970.

`kern_return_t host_get_time` (*host_t* `host`, *time_value_t* `*current_time`) [Function]
: Get the current time as seen by *host*. On success, the time passed since the epoch is returned in *current_time*.

`kern_return_t host_set_time` (*host_priv_t* `host_priv`, *time_value_t* `new_time`) [Function]
: Set the time of *host_priv* to *new_time*.

`kern_return_t host_adjust_time` (*host_priv_t* `host_priv`, *time_value_t* `new_adjustment`, *time_value_t* `*old_adjustment`) [Function]
: Arrange for the current time as seen by *host_priv* to be gradually changed by the adjustment value *new_adjustment*, and return the old adjustment value in *old_adjustment*.

For efficiency, the current time is available through a mapped-time interface.

`mapped_time_value_t` [Data type]
: This structure defines the mapped-time interface. It has the following members:

  `integer_t seconds`
  : The number of seconds.

  `integer_t microseconds`
  : The number of microseconds.

  `integer_t check_seconds`
  : This is a copy of the seconds value, which must be checked to protect against a race condition when reading out the two time values.

Here is an example how to read out the current time using the mapped-time interface:

```
do
  {
    secs = mtime->seconds;
    usecs = mtime->microseconds;
  }
while (secs != mtime->check_seconds);
```

8.4 Host Reboot

kern_return_t host_reboot (*host_priv_t* host_priv, *int* options) [Function]

Reboot the host specified by *host_priv*. The argument *options* specifies the flags. The available flags are defined in 'sys/reboot.h':

RB_HALT
: Do not reboot, but halt the machine.

RB_DEBUGGER
: Do not reboot, but enter kernel debugger from user space.

If successful, the function might not return.

9 Processors and Processor Sets

This section describes the Mach interface to processor sets and individual processors. The interface allows to group processors into sets and control the processors and processor sets.

A processor is not a central part of the interface. It is mostly of relevance as a part of a processor set. Threads are always assigned to processor sets, and all processors in a set are equally involved in executing all threads assigned to that set.

The processor set is represented by two ports, a name port *processor_set_name* used to query information about the host accessible to everyone, and a control port *processor_set* used to manipulate it.

9.1 Processor Set Interface

9.1.1 Processor Set Ports

`processor_set_name_t` [Data type]

This is a `mach_port_t` and used to hold the port name of a processor set name port that names the processor set. Any task can get a send right to name port of a processor set. The processor set name port allows to get information about the processor set.

`processor_set_t` [Data type]

This is a `mach_port_t` and used to hold the port name of a privileged processor set control port that represents the processor set. Operations on the processor set are implemented as remote procedure calls to the processor set port. The processor set port allows to manipulate the processor set.

9.1.2 Processor Set Access

`kern_return_t host_processor_sets` (*host_t* `host`, *processor_set_name_array_t* *`processor_sets`, *mach_msg_type_number_t* *`processor_sets_count`) [Function]

The function `host_processor_sets` gets send rights to the name port for each processor set currently assigned to *host*.

`host_processor_set_priv` can be used to obtain the control ports from these if desired. *processor_sets* is an array that is created as a result of this call. The caller may wish to `vm_deallocate` this array when the data is no longer needed. *processor_sets_count* is set to the number of processor sets in the *processor_sets*.

This function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *host* is not a host.

`kern_return_t host_processor_set_priv` (*host_priv_t* `host_priv`, *processor_set_name_t* `set_name`, *processor_set_t* *`set`) [Function]

The function `host_processor_set_priv` allows a privileged application to obtain the control port *set* for an existing processor set from its name port *set_name*. The privileged host port *host_priv* is required.

This function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *host_priv* is not a valid host control port.

kern_return_t processor_set_default (*host_t* host, *processor_set_name_t* *default_set) [Function]

The function processor_set_default returns the default processor set of *host* in *default_set*. The default processor set is used by all threads, tasks, and processors that are not explicitly assigned to other sets. processor_set_default returns a port that can be used to obtain information about this set (e.g. how many threads are assigned to it). This port cannot be used to perform operations on that set.

This function returns KERN_SUCCESS if the call succeeded, KERN_INVALID_ARGUMENT if *host* is not a host and KERN_INVALID_ADDRESS if *default_set* points to inaccessible memory.

9.1.3 Processor Set Creation

kern_return_t processor_set_create (*host_t* host, *processor_set_t* *new_set, *processor_set_name_t* *new_name) [Function]

The function processor_set_create creates a new processor set on *host* and returns the two ports associated with it. The port returned in *new_set* is the actual port representing the set. It is used to perform operations such as assigning processors, tasks, or threads. The port returned in *new_name* identifies the set, and is used to obtain information about the set.

This function returns KERN_SUCCESS if the call succeeded, KERN_INVALID_ARGUMENT if *host* is not a host, KERN_INVALID_ADDRESS if *new_set* or *new_name* points to inaccessible memory and KERN_FAILURE is the operating system does not support processor allocation.

9.1.4 Processor Set Destruction

kern_return_t processor_set_destroy (*processor_set_t* processor_set) [Function]

The function processor_set_destroy destroys the specified processor set. Any assigned processors, tasks, or threads are reassigned to the default set. The object port for the processor set is required (not the name port). The default processor set cannot be destroyed.

This function returns KERN_SUCCESS if the set was destroyed, KERN_FAILURE if an attempt was made to destroy the default processor set, or the operating system does not support processor allocation, and KERN_INVALID_ARGUMENT if *processor_set* is not a valid processor set control port.

9.1.5 Tasks and Threads on Sets

kern_return_t processor_set_tasks (*processor_set_t* processor_set, *task_array_t* *task_list, *mach_msg_type_number_t* *task_count) [Function]

The function processor_set_tasks gets send rights to the kernel port for each task currently assigned to *processor_set*.

task_list is an array that is created as a result of this call. The caller may wish to vm_deallocate this array when the data is no longer needed. *task_count* is set to the number of tasks in the *task_list*.

This function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *processor_set* is not a processor set.

`kern_return_t processor_set_threads` [Function]
(*processor_set_t* `processor_set`, *thread_array_t* `*thread_list`, *mach_msg_type_number_t* `*thread_count`)

The function `processor_set_thread` gets send rights to the kernel port for each thread currently assigned to *processor_set*.

thread_list is an array that is created as a result of this call. The caller may wish to `vm_deallocate` this array when the data is no longer needed. *thread_count* is set to the number of threads in the *thread_list*.

This function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *processor_set* is not a processor set.

`kern_return_t task_assign` (*task_t* `task`, [Function]
processor_set_t `processor_set`, *boolean_t* `assign_threads`)

The function `task_assign` assigns *task* the set *processor_set*. This assignment is for the purposes of determining the initial assignment of newly created threads in task. Any previous assignment of the task is nullified. Existing threads within the task are also reassigned if *assign_threads* is `TRUE`. They are not affected if it is `FALSE`.

This function returns `KERN_SUCCESS` if the assignment has been performed and `KERN_INVALID_ARGUMENT` if *task* is not a task, or *processor_set* is not a processor set on the same host as *task*.

`kern_return_t task_assign_default` (*task_t* `task`, [Function]
boolean_t `assign_threads`)

The function `task_assign_default` is a variant of `task_assign` that assigns the task to the default processor set on that task's host. This variant exists because the control port for the default processor set is privileged and not ususally available to users.

This function returns `KERN_SUCCESS` if the assignment has been performed and `KERN_INVALID_ARGUMENT` if *task* is not a task.

`kern_return_t task_get_assignment` (*task_t* `task`, [Function]
processor_set_name_t `*assigned_set`)

The function `task_get_assignment` returns the name of the processor set to which the thread is currently assigned in *assigned_set*. This port can only be used to obtain information about the processor set.

This function returns `KERN_SUCCESS` if the assignment has been performed, `KERN_INVALID_ADDRESS` if *processor_set* points to inaccessible memory, and `KERN_INVALID_ARGUMENT` if *task* is not a task.

`kern_return_t thread_assign` (*thread_t* `thread`, [Function]
processor_set_t `processor_set`)

The function `thread_assign` assigns *thread* the set *processor_set*. After the assignment is completed, the thread only executes on processors assigned to the designated processor set. If there are no such processors, then the thread is unable to execute.

Any previous assignment of the thread is nullified. Unix system call compatibility code may temporarily force threads to execute on the master processor.

This function returns `KERN_SUCCESS` if the assignment has been performed and `KERN_INVALID_ARGUMENT` if *thread* is not a thread, or *processor_set* is not a processor set on the same host as *thread*.

`kern_return_t thread_assign_default` (*thread_t* `thread`) [Function]

The function `thread_assign_default` is a variant of `thread_assign` that assigns the thread to the default processor set on that thread's host. This variant exists because the control port for the default processor set is privileged and not ususally available to users.

This function returns `KERN_SUCCESS` if the assignment has been performed and `KERN_INVALID_ARGUMENT` if *thread* is not a thread.

`kern_return_t thread_get_assignment` (*thread_t* `thread`, *processor_set_name_t* *`assigned_set`) [Function]

The function `thread_get_assignment` returns the name of the processor set to which the thread is currently assigned in *assigned_set*. This port can only be used to obtain information about the processor set.

This function returns `KERN_SUCCESS` if the assignment has been performed, `KERN_INVALID_ADDRESS` if *processor_set* points to inaccessible memory, and `KERN_INVALID_ARGUMENT` if *thread* is not a thread.

9.1.6 Processor Set Priority

`kern_return_t processor_set_max_priority` (*processor_set_t* `processor_set`, *int* `max_priority`, *boolean_t* `change_threads`) [Function]

The function `processor_set_max_priority` is used to set the maximum priority for a processor set. The priority of a processor set is used only for newly created threads (thread's maximum priority is set to processor set's) and the assignment of threads to the set (thread's maximum priority is reduced if it exceeds the set's maximum priority, thread's priority is similarly reduced). `processor_set_max_priority` changes this priority. It also sets the maximum priority of all threads assigned to the processor set to this new priority if *change_threads* is `TRUE`. If this maximum priority is less than the priorities of any of these threads, their priorities will also be set to this new value.

This function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *processor_set* is not a processor set or *priority* is not a valid priority.

9.1.7 Processor Set Policy

`kern_return_t processor_set_policy_enable` (*processor_set_t* `processor_set`, *int* `policy`) [Function]

`kern_return_t processor_set_policy_disable` [Function]
(*processor_set_t* `processor_set`, *int* `policy`, *boolean_t* `change_threads`)

Processor sets may restrict the scheduling policies to be used for threads assigned to them. These two calls provide the mechanism for designating permitted and forbidden policies. The current set of permitted policies can be obtained from `processor_set_info`. Timesharing may not be forbidden by any processor set. This is a compromise to reduce the complexity of the assign operation; any thread whose policy is forbidden by the target processor set has its policy reset to timesharing. If the *change_threads* argument to `processor_set_policy_disable` is true, threads currently assigned to this processor set and using the newly disabled policy will have their policy reset to timesharing.

'`mach/policy.h`' contains the allowed policies; it is included by '`mach.h`'. Not all policies (e.g. fixed priority) are supported by all systems.

This function returns `KERN_SUCCESS` if the operation was completed successfully and `KERN_INVALID_ARGUMENT` if *processor_set* is not a processor set or *policy* is not a valid policy, or an attempt was made to disable timesharing.

9.1.8 Processor Set Info

`kern_return_t processor_set_info` [Function]
(*processor_set_name_t* `set_name`, *int* `flavor`, *host_t* `*host`, *processor_set_info_t* `processor_set_info`, *mach_msg_type_number_t* `*processor_set_info_count`)

The function `processor_set_info` returns the selected information array for a processor set, as specified by *flavor*.

host is set to the host on which the processor set resides. This is the non-privileged host port.

processor_set_info is an array of integers that is supplied by the caller and returned filled with specified information. *processor_set_info_count* is supplied as the maximum number of integers in *processor_set_info*. On return, it contains the actual number of integers in *processor_set_info*. The maximum number of integers returned by any flavor is `PROCESSOR_SET_INFO_MAX`.

The type of information returned is defined by *flavor*, which can be one of the following:

`PROCESSOR_SET_BASIC_INFO`
: The function returns basic information about the processor set, as defined by `processor_set_basic_info_t`. This includes the number of tasks and threads assigned to the processor set. The number of integers returned is `PROCESSOR_SET_BASIC_INFO_COUNT`.

`PROCESSOR_SET_SCHED_INFO`
: The function returns information about the schduling policy for the processor set as defined by `processor_set_sched_info_t`. The number of integers returned is `PROCESSOR_SET_SCHED_INFO_COUNT`.

Some machines may define additional (machine-dependent) flavors.

The function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *processor_set* is not a processor set or *flavor* is not recognized. The function returns `MIG_ARRAY_TOO_LARGE` if the returned info array is too large for *processor_set_info*. In this case, *processor_set_info* is filled as much as possible and *processor_set_info_count* is set to the number of elements that would have been returned if there were enough room.

`struct processor_set_basic_info` [Data type]

This structure is returned in *processor_set_info* by the `processor_set_info` function and provides basic information about the processor set. You can cast a variable of type `processor_set_info_t` to a pointer of this type if you provided it as the *processor_set_info* parameter for the `PROCESSOR_SET_BASIC_INFO` flavor of `processor_set_info`. It has the following members:

`int processor_count`
: number of processors

`int task_count`
: number of tasks

`int thread_count`
: number of threads

`int load_average`
: scaled load average

`int mach_factor`
: scaled mach factor

`processor_set_basic_info_t` [Data type]

This is a pointer to a `struct processor_set_basic_info`.

`struct processor_set_sched_info` [Data type]

This structure is returned in *processor_set_info* by the `processor_set_info` function and provides schedule information about the processor set. You can cast a variable of type `processor_set_info_t` to a pointer of this type if you provided it as the *processor_set_info* parameter for the `PROCESSOR_SET_SCHED_INFO` flavor of `processor_set_info`. It has the following members:

`int policies`
: allowed policies

`int max_priority`
: max priority for new threads

`processor_set_sched_info_t` [Data type]

This is a pointer to a `struct processor_set_sched_info`.

9.2 Processor Interface

`processor_t` [Data type]

This is a `mach_port_t` and used to hold the port name of a processor port that represents the processor. Operations on the processor are implemented as remote procedure calls to the processor port.

9.2.1 Hosted Processors

kern_return_t **host_processors** (*host_priv_t* host_priv, *processor_array_t* *processor_list, *mach_msg_type_number_t* *processor_count) [Function]

The function host_processors gets send rights to the processor port for each processor existing on *host_priv*. This is the privileged port that allows its holder to control a processor.

processor_list is an array that is created as a result of this call. The caller may wish to vm_deallocate this array when the data is no longer needed. *processor_count* is set to the number of processors in the *processor_list*.

This function returns KERN_SUCCESS if the call succeeded, KERN_INVALID_ARGUMENT if *host_priv* is not a privileged host port, and KERN_INVALID_ADDRESS if *processor_count* points to inaccessible memory.

9.2.2 Processor Control

kern_return_t **processor_start** (*processor_t* processor) [Function]

kern_return_t **processor_exit** (*processor_t* processor) [Function]

kern_return_t **processor_control** (*processor_t* processor, *processor_info_t* *cmd, *mach_msg_type_number_t* count) [Function]

Some multiprocessors may allow privileged software to control processors. The processor_start, processor_exit, and processor_control operations implement this. The interpretation of the command in *cmd* is machine dependent. A newly started processor is assigned to the default processor set. An exited processor is removed from the processor set to which it was assigned and ceases to be active.

count contains the length of the command *cmd* as a number of ints.

Availability limited. All of these operations are machine-dependent. They may do nothing. The ability to restart an exited processor is also machine-dependent.

This function returns KERN_SUCCESS if the operation was performed, KERN_FAILURE if the operation was not performed (a likely reason is that it is not supported on this processor), KERN_INVALID_ARGUMENT if *processor* is not a processor, and KERN_INVALID_ADDRESS if *cmd* points to inaccessible memory.

9.2.3 Processors and Sets

kern_return_t **processor_assign** (*processor_t* processor, *processor_set_t* processor_set, *boolean_t* wait) [Function]

The function processor_assign assigns *processor* to the the set *processor_set*. After the assignment is completed, the processor only executes threads that are assigned to that processor set. Any previous assignment of the processor is nullified. The master processor cannot be reassigned. All processors take clock interrupts at all times. The *wait* argument indicates whether the caller should wait for the assignment to be completed or should return immediately. Dedicated kernel threads are used to perform processor assignment, so setting wait to FALSE allows assignment requests to be queued and performed faster, especially if the kernel has more than one dedicated internal thread for processor assignment. Redirection of other device interrupts away from processors assigned to other than the default processor set is machine-dependent. Intermediaries that interpose on ports must be sure to interpose on both ports involved in this call if they interpose on either.

This function returns `KERN_SUCCESS` if the assignment has been performed, `KERN_INVALID_ARGUMENT` if *processor* is not a processor, or *processor_set* is not a processor set on the same host as *processor*.

`kern_return_t processor_get_assignment` (*processor_t* `processor`, *processor_set_name_t* *`assigned_set`) [Function]

The function `processor_get_assignment` obtains the current assignment of a processor. The name port of the processor set is returned in *assigned_set*.

9.2.4 Processor Info

`kern_return_t processor_info` (*processor_t* `processor`, *int* `flavor`, *host_t* *`host`, *processor_info_t* `processor_info`, *mach_msg_type_number_t* *`processor_info_count`) [Function]

The function `processor_info` returns the selected information array for a processor, as specified by *flavor*.

host is set to the host on which the processor set resides. This is the non-privileged host port.

processor_info is an array of integers that is supplied by the caller and returned filled with specified information. *processor_info_count* is supplied as the maximum number of integers in *processor_info*. On return, it contains the actual number of integers in *processor_info*. The maximum number of integers returned by any flavor is `PROCESSOR_INFO_MAX`.

The type of information returned is defined by *flavor*, which can be one of the following:

`PROCESSOR_BASIC_INFO`
: The function returns basic information about the processor, as defined by `processor_basic_info_t`. This includes the slot number of the processor. The number of integers returned is `PROCESSOR_BASIC_INFO_COUNT`.

Machines which require more configuration information beyond the slot number are expected to define additional (machine-dependent) flavors.

The function returns `KERN_SUCCESS` if the call succeeded and `KERN_INVALID_ARGUMENT` if *processor* is not a processor or *flavor* is not recognized. The function returns `MIG_ARRAY_TOO_LARGE` if the returned info array is too large for *processor_info*. In this case, *processor_info* is filled as much as possible and *processor_infoCnt* is set to the number of elements that would have been returned if there were enough room.

`struct processor_basic_info` [Data type]

This structure is returned in *processor_info* by the `processor_info` function and provides basic information about the processor. You can cast a variable of type `processor_info_t` to a pointer of this type if you provided it as the *processor_info* parameter for the `PROCESSOR_BASIC_INFO` flavor of `processor_info`. It has the following members:

`cpu_type_t cpu_type`
: cpu type

`cpu_subtype_t cpu_subtype`
: cpu subtype

`boolean_t running`
: is processor running?

`int slot_num`
: slot number

`boolean_t is_master`
: is this the master processor

`processor_basic_info_t` [Data type]

This is a pointer to a `struct processor_basic_info`.

10 Device Interface

The GNU Mach microkernel provides a simple device interface that allows the user space programs to access the underlying hardware devices. Each device has a unique name, which is a string up to 127 characters long. To open a device, the device master port has to be supplied. The device master port is only available through the bootstrap port. Anyone who has control over the device master port can use all hardware devices.

`device_t` [Data type]

This is a `mach_port_t` and used to hold the port name of a device port that represents the device. Operations on the device are implemented as remote procedure calls to the device port. Each device provides a sequence of records. The length of a record is specific to the device. Data can be transferred "out-of-line" or "in-line" (see Section 4.2.4 [Memory], page 21).

All constants and functions in this chapter are defined in '`device/device.h`'.

10.1 Device Reply Server

Beside the usual synchronous interface, an asynchronous interface is provided. For this, the caller has to receive and handle the reply messages seperately from the function call.

`boolean_t device_reply_server` (*msg_header_t* `*in_msg`, *msg_header_t* `*out_msg`) [Function]

The function `device_reply_server` is produced by the remote procedure call generator to handle a received message. This function does all necessary argument handling, and actually calls one of the following functions: `ds_device_open_reply`, `ds_device_read_reply`, `ds_device_read_reply_inband`, `ds_device_write_reply` and `ds_device_write_reply_inband`.

The *in_msg* argument is the message that has been received from the kernel. The *out_msg* is a reply message, but this is not used for this server.

The function returns `TRUE` to indicate that the message in question was applicable to this interface, and that the appropriate routine was called to interpret the message. It returns `FALSE` to indicate that the message did not apply to this interface, and that no other action was taken.

10.2 Device Open

`kern_return_t device_open` (*mach_port_t* `master_port`, *dev_mode_t* `mode`, *dev_name_t* `name`, *device_t* `*device`) [Function]

The function `device_open` opens the device *name* and returns a port to it in *device*. The open count for the device is incremented by one. If the open count was 0, the open handler for the device is invoked.

master_port must hold the master device port. *name* specifies the device to open, and is a string up to 128 characters long. *mode* is the open mode. It is a bitwise-or of the following constants:

`D_READ` Request read access for the device.

`D_WRITE` Request write access for the device.

`D_NODELAY`
Do not delay an open.

The function returns `D_SUCCESS` if the device was successfully opened, `D_INVALID_OPERATION` if *master_port* is not the master device port, `D_WOULD_BLOCK` is the device is busy and `D_NOWAIT` was specified in mode, `D_ALREADY_OPEN` if the device is already open in an incompatible mode and `D_NO_SUCH_DEVICE` if *name* does not denote a know device.

`kern_return_t device_open_request` (*mach_port_t* `master_port`, *mach_port_t* `reply_port`, *dev_mode_t* `mode`, *dev_name_t* `name`) [Function]

`kern_return_t ds_device_open_reply` (*mach_port_t* `reply_port`, *kern_return_t* `return`, *device_t* `*device`) [Function]

This is the asynchronous form of the `device_open` function. `device_open_request` performs the open request. The meaning for the parameters is as in `device_open`. Additionally, the caller has to supply a reply port to which the `ds_device_open_reply` message is sent by the kernel when the open has been performed. The return value of the open operation is stored in *return_code*.

As neither function receives a reply message, only message transmission errors apply. If no error occurs, `KERN_SUCCESS` is returned.

10.3 Device Close

`kern_return_t device_close` (*device_t* `device`) [Function]

The function `device_close` decrements the open count of the device by one. If the open count drops to zero, the close handler for the device is called. The device to close is specified by its port *device*.

The function returns `D_SUCCESS` if the device was successfully closed and `D_NO_SUCH_DEVICE` if *device* does not denote a device port.

10.4 Device Read

`kern_return_t device_read` (*device_t* `device`, *dev_mode_t* `mode`, *recnum_t* `recnum`, *int* `bytes_wanted`, *io_buf_ptr_t* `*data`, *mach_msg_type_number_t* `*data_count`) [Function]

The function `device_read` reads *bytes_wanted* bytes from *device*, and stores them in a buffer allocated with `vm_allocate`, which address is returned in *data*. The caller must deallocated it if it is no longer needed. The number of bytes actually returned is stored in *data_count*.

If *mode* is `D_NOWAIT`, the operation does not block. Otherwise *mode* should be 0. *recnum* is the record number to be read, its meaning is device specific.

The function returns `D_SUCCESS` if some data was successfully read, `D_WOULD_BLOCK` if no data is currently available and `D_NOWAIT` is specified, and `D_NO_SUCH_DEVICE` if *device* does not denote a device port.

kern_return_t device_read_inband (*device_t* device, *dev_mode_t* mode, *recnum_t* recnum, *int* bytes_wanted, *io_buf_ptr_inband_t* *data, *mach_msg_type_number_t* *data_count) [Function]

The device_read_inband function works as the device_read function, except that the data is returned "in-line" in the reply IPC message (see Section 4.2.4 [Memory], page 21).

kern_return_t device_read_request (*device_t* device, *mach_port_t* reply_port, *dev_mode_t* mode, *recnum_t* recnum, *int* bytes_wanted) [Function]

kern_return_t ds_device_read_reply (*mach_port_t* reply_port, *kern_return_t* return_code, *io_buf_ptr_t* data, *mach_msg_type_number_t* data_count) [Function]

This is the asynchronous form of the device_read function. device_read_request performs the read request. The meaning for the parameters is as in device_read. Additionally, the caller has to supply a reply port to which the ds_device_read_reply message is sent by the kernel when the read has been performed. The return value of the read operation is stored in *return_code*.

As neither function receives a reply message, only message transmission errors apply. If no error occurs, KERN_SUCCESS is returned.

kern_return_t device_read_request_inband (*device_t* device, *mach_port_t* reply_port, *dev_mode_t* mode, *recnum_t* recnum, *int* bytes_wanted) [Function]

kern_return_t ds_device_read_reply_inband (*mach_port_t* reply_port, *kern_return_t* return_code, *io_buf_ptr_t* data, *mach_msg_type_number_t* data_count) [Function]

The device_read_request_inband and ds_device_read_reply_inband functions work as the device_read_request and ds_device_read_reply functions, except that the data is returned "in-line" in the reply IPC message (see Section 4.2.4 [Memory], page 21).

10.5 Device Write

kern_return_t device_write (*device_t* device, *dev_mode_t* mode, *recnum_t* recnum, *io_buf_ptr_t* data, *mach_msg_type_number_t* data_count, *int* *bytes_written) [Function]

The function device_write writes *data_count* bytes from the buffer *data* to *device*. The number of bytes actually written is returned in *bytes_written*.

If *mode* is D_NOWAIT, the function returns without waiting for I/O completion. Otherwise *mode* should be 0. *recnum* is the record number to be written, its meaning is device specific.

The function returns D_SUCCESS if some data was successfully written and D_NO_SUCH_DEVICE if *device* does not denote a device port or the device is dead or not completely open.

kern_return_t device_write_inband (*device_t* device, *dev_mode_t* mode, *recnum_t* recnum, *int* bytes_wanted, *io_buf_ptr_inband_t* *data, *mach_msg_type_number_t* *data_count) [Function]

The `device_write_inband` function works as the `device_write` function, except that the data is sent "in-line" in the request IPC message (see Section 4.2.4 [Memory], page 21).

kern_return_t device_write_request (*device_t* device, *mach_port_t* reply_port, *dev_mode_t* mode, *recnum_t* recnum, *io_buf_ptr_t* data, *mach_msg_type_number_t* data_count) [Function]

kern_return_t ds_device_write_reply (*mach_port_t* reply_port, *kern_return_t* return_code, *int* bytes_written) [Function]

This is the asynchronous form of the `device_write` function. `device_write_request` performs the write request. The meaning for the parameters is as in `device_write`. Additionally, the caller has to supply a reply port to which the `ds_device_write_reply` message is sent by the kernel when the write has been performed. The return value of the write operation is stored in *return_code*.

As neither function receives a reply message, only message transmission errors apply. If no error occurs, `KERN_SUCCESS` is returned.

kern_return_t device_write_request_inband (*device_t* device, *mach_port_t* reply_port, *dev_mode_t* mode, *recnum_t* recnum, *io_buf_ptr_t* data, *mach_msg_type_number_t* data_count) [Function]

kern_return_t ds_device_write_reply_inband (*mach_port_t* reply_port, *kern_return_t* return_code, *int* bytes_written) [Function]

The `device_write_request_inband` and `ds_device_write_reply_inband` functions work as the `device_write_request` and `ds_device_write_reply` functions, except that the data is sent "in-line" in the request IPC message (see Section 4.2.4 [Memory], page 21).

10.6 Device Map

kern_return_t device_map (*device_t* device, *vm_prot_t* prot, *vm_offset_t* offset, *vm_size_t* size, *mach_port_t* *pager, *int* unmap) [Function]

The function `device_map` creates a new memory manager for *device* and returns a port to it in *pager*. The memory manager is usable as a memory object in a `vm_map` call. The call is device dependant.

The protection for the memory object is specified by *prot*. The memory object starts at *offset* within the device and extends *size* bytes. *unmap* is currently unused.

The function returns `D_SUCCESS` if some data was successfully written and `D_NO_SUCH_DEVICE` if *device* does not denote a device port or the device is dead or not completely open.

10.7 Device Status

kern_return_t **device_set_status** (*device_t* **device**, *dev_flavor_t* **flavor**, *dev_status_t* **status**, *mach_msg_type_number_t* **status_count**) [Function]

The function `device_set_status` sets the status of a device. The possible values for *flavor* and their interpretation is device specific.

The function returns `D_SUCCESS` if some data was successfully written and `D_NO_SUCH_DEVICE` if *device* does not denote a device port or the device is dead or not completely open.

kern_return_t **device_get_status** (*device_t* **device**, *dev_flavor_t* **flavor**, *dev_status_t* **status**, *mach_msg_type_number_t* ***status_count**) [Function]

The function `device_get_status` gets the status of a device. The possible values for *flavor* and their interpretation is device specific.

The function returns `D_SUCCESS` if some data was successfully written and `D_NO_SUCH_DEVICE` if *device* does not denote a device port or the device is dead or not completely open.

10.8 Device Filter

kern_return_t **device_set_filter** (*device_t* **device**, *mach_port_t* **receive_port**, *mach_msg_type_name_t* **receive_port_type**, *int* **priority**, *filter_array_t* **filter**, *mach_msg_type_number_t* **filter_count**) [Function]

The function `device_set_filter` makes it possible to filter out selected data arriving at the device and forward it to a port. *filter* is a list of filter commands, which are applied to incoming data to determine if the data should be sent to *receive_port*. The IPC type of the send right is specified by *receive_port_right*, it is either `MACH_MSG_TYPE_MAKE_SEND` or `MACH_MSG_TYPE_MOVE_SEND`. The *priority* value is used to order multiple filters.

There can be up to `NET_MAX_FILTER` commands in *filter*. The actual number of commands is passed in *filter_count*. For the purpose of the filter test, an internal stack is provided. After all commands have been processed, the value on the top of the stack determines if the data is forwarded or the next filter is tried.

Each word of the command list specifies a data (push) operation (high order NETF_NBPO bits) as well as a binary operator (low order NETF_NBPA bits). The value to be pushed onto the stack is chosen as follows.

`NETF_PUSHLIT`
: Use the next short word of the filter as the value.

`NETF_PUSHZERO`
: Use 0 as the value.

`NETF_PUSHWORD+N`
: Use short word N of the "data" portion of the message as the value.

`NETF_PUSHHDR+N`
: Use short word N of the "header" portion of the message as the value.

`NETF_PUSHIND+N`
: Pops the top long word from the stack and then uses short word N of the "data" portion of the message as the value.

`NETF_PUSHHDRIND+N`
: Pops the top long word from the stack and then uses short word N of the "header" portion of the message as the value.

`NETF_PUSHSTK+N`
: Use long word N of the stack (where the top of stack is long word 0) as the value.

`NETF_NOPUSH`
: Don't push a value.

The unsigned value so chosen is promoted to a long word before being pushed. Once a value is pushed (except for the case of `NETF_NOPUSH`), the top two long words of the stack are popped and a binary operator applied to them (with the old top of stack as the second operand). The result of the operator is pushed on the stack. These operators are:

`NETF_NOP` Don't pop off any values and do no operation.

`NETF_EQ` Perform an equal comparison.

`NETF_LT` Perform a less than comparison.

`NETF_LE` Perform a less than or equal comparison.

`NETF_GT` Perform a greater than comparison.

`NETF_GE` Perform a greater than or equal comparison.

`NETF_AND` Perform a bitise boolean AND operation.

`NETF_OR` Perform a bitise boolean inclusive OR operation.

`NETF_XOR` Perform a bitise boolean exclusive OR operation.

`NETF_NEQ` Perform a not equal comparison.

`NETF_LSH` Perform a left shift operation.

`NETF_RSH` Perform a right shift operation.

`NETF_ADD` Perform an addition.

`NETF_SUB` Perform a subtraction.

`NETF_COR` Perform an equal comparison. If the comparison is `TRUE`, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

`NETF_CAND`
: Perform an equal comparison. If the comparison is `FALSE`, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

`NETF_CNOR`
: Perform a not equal comparison. If the comparison is `FALSE`, terminate the filter list. Otherwise, pop the result of the comparison off the stack.

`NETF_CNAND`
: Perform a not equal comparison. If the comparison is `TRUE`, terminate the filter list. Otherwise, pop the result of the comparison off the stack. The scan of the filter list terminates when the filter list is emptied, or a `NETF_C...` operation terminates the list. At this time, if the final value of the top of the stack is `TRUE`, then the message is accepted for the filter.

The function returns `D_SUCCESS` if some data was successfully written, `D_INVALID_OPERATION` if *receive_port* is not a valid send right, and `D_NO_SUCH_DEVICE` if *device* does not denote a device port or the device is dead or not completely open.

11 Kernel Debugger

The GNU Mach kernel debugger `ddb` is a powerful built-in debugger with a gdb like syntax. It is enabled at compile time using the '`--enable-kdb`' option. Whenever you want to enter the debugger while running the kernel, you can press the key combination ⟨Ctrl-Alt-D⟩.

11.1 Operation

The current location is called *dot*. The dot is displayed with a hexadecimal format at a prompt. Examine and write commands update dot to the address of the last line examined or the last location modified, and set *next* to the address of the next location to be examined or changed. Other commands don't change dot, and set next to be the same as dot.

The general command syntax is:

```
command[/modifier] address [,count]
```

`!!` repeats the previous command, and a blank line repeats from the address next with count 1 and no modifiers. Specifying *address* sets dot to the address. Omitting *address* uses dot. A missing *count* is taken to be 1 for printing commands or infinity for stack traces.

Current `ddb` is enhanced to support multi-thread debugging. A break point can be set only for a specific thread, and the address space or registers of non current thread can be examined or modified if supported by machine dependent routines. For example,

```
break/t mach_msg_trap $task11.0
```

sets a break point at `mach_msg_trap` for the first thread of task 11 listed by a `show all threads` command.

In the above example, `$task11.0` is translated to the corresponding thread structure's address by variable translation mechanism described later. If a default target thread is set in a variable `$thread`, the `$task11.0` can be omitted. In general, if `t` is specified in a modifier of a command line, a specified thread or a default target thread is used as a target thread instead of the current one. The `t` modifier in a command line is not valid in evaluating expressions in a command line. If you want to get a value indirectly from a specific thread's address space or access to its registers within an expression, you have to specify a default target thread in advance, and to use `:t` modifier immediately after the indirect access or the register reference like as follows:

```
set $thread $task11.0
print $eax:t *(0x100):tuh
```

No sign extension and indirection `size(long, half word, byte)` can be specified with `u`, `l`, `h` and `b` respectively for the indirect access.

Note: Support of non current space/register access and user space break point depend on the machines. If not supported, attempts of such operation may provide incorrect information or may cause strange behavior. Even if supported, the user space access is limited to the pages resident in the main memory at that time. If a target page is not in the main memory, an error will be reported.

`ddb` has a feature like a command `more` for the output. If an output line exceeds the number set in the `$lines` variable, it displays '`--db_more--`' and waits for a response. The valid responses for it are:

⟨SPC⟩ one more page

⟨RET⟩ one more line

`q` abort the current command, and return to the command input mode

11.2 Commands

`examine(x) [/modifier] addr[,count] [ thread ]`

Display the addressed locations according to the formats in the modifier. Multiple modifier formats display multiple locations. If no format is specified, the last formats specified for this command is used. Address space other than that of the current thread can be specified with `t` option in the modifier and *thread* parameter. The format characters are

- `b` look at by bytes(8 bits)
- `h` look at by half words(16 bits)
- `l` look at by long words(32 bits)
- `a` print the location being displayed
- `,` skip one unit producing no output
- `A` print the location with a line number if possible
- `x` display in unsigned hex
- `z` display in signed hex
- `o` display in unsigned octal
- `d` display in signed decimal
- `u` display in unsigned decimal
- `r` display in current radix, signed
- `c` display low 8 bits as a character. Non-printing characters are displayed as an octal escape code (e.g. '\000').
- `s` display the null-terminated string at the location. Non-printing characters are displayed as octal escapes.
- `m` display in unsigned hex with character dump at the end of each line. The location is also displayed in hex at the beginning of each line.
- `i` display as an instruction
- `I` display as an instruction with possible alternate formats depending on the machine:
  - `vax` don't assume that each external label is a procedure entry mask
  - `i386` don't round to the next long word boundary
  - `mips` print register contents

`xf` Examine forward. It executes an examine command with the last specified parameters to it except that the next address displayed by it is used as the start address.

`xb` Examine backward. It executes an examine command with the last specified parameters to it except that the last start address subtracted by the size displayed by it is used as the start address.

`print[/axzodurc]` *`addr1`* `[` *`addr2`* `... ]`

Print *addr*'s according to the modifier character. Valid formats are: `a x z o d u r c`. If no modifier is specified, the last one specified to it is used. *addr* can be a string, and it is printed as it is. For example,

```
print/x "eax = " $eax "\necx = " $ecx "\n"
```

will print like

```
eax = xxxxxx
ecx = yyyyyy
```

`write[/bhlt]` *`addr`* `[` *`thread`* `]` *`expr1`* `[` *`expr2`* `... ]`

Write the expressions at succeeding locations. The write unit size can be specified in the modifier with a letter b (byte), h (half word) or l(long word) respectively. If omitted, long word is assumed. Target address space can also be specified with `t` option in the modifier and *thread* parameter. Warning: since there is no delimiter between expressions, strange things may happen. It's best to enclose each expression in parentheses.

`set $`*`variable`* `[=]` *`expr`*

Set the named variable or register with the value of *expr*. Valid variable names are described below.

`break[/tuTU]` *`addr`*`[,`*`count`*`] [` *`thread1`* `... ]`

Set a break point at *addr*. If count is supplied, continues (*count*-1) times before stopping at the break point. If the break point is set, a break point number is printed with '`#`'. This number can be used in deleting the break point or adding conditions to it.

`t` Set a break point only for a specific thread. The thread is specified by *thread* parameter, or default one is used if the parameter is omitted.

`u` Set a break point in user space address. It may be combined with `t` or `T` option to specify the non-current target user space. Without `u` option, the address is considered in the kernel space, and wrong space address is rejected with an error message. This option can be used only if it is supported by machine dependent routines.

`T` Set a break point only for threads in a specific task. It is like `t` option except that the break point is valid for all threads which belong to the same task as the specified target thread.

`U` Set a break point in shared user space address. It is like `u` option, except that the break point is valid for all threads which share the same address space even if `t` option is specified. `t` option is used only to specify the target shared space. Without `t` option, `u` and `U` have the same meanings. `U` is useful for setting a user space break point in non-current address space with `t` option such as in an emulation library space. This option can be used only if it is supported by machine dependent routines.

Warning: if a user text is shadowed by a normal user space debugger, user space break points may not work correctly. Setting a break point at the low-level code paths may also cause strange behavior.

`delete[/tuTU] addr|#number [ thread1 ... ]`
: Delete the break point. The target break point can be specified by a break point number with `#`, or by *addr* like specified in `break` command.

`cond #number [ condition commands ]`
: Set or delete a condition for the break point specified by the *number*. If the *condition* and *commands* are null, the condition is deleted. Otherwise the condition is set for it. When the break point is hit, the *condition* is evaluated. The *commands* will be executed if the condition is true and the break point count set by a break point command becomes zero. *commands* is a list of commands separated by semicolons. Each command in the list is executed in that order, but if a `continue` command is executed, the command execution stops there, and the stopped thread resumes execution. If the command execution reaches the end of the list, and it enters into a command input mode. For example,

```
set $work0 0
break/Tu xxx_start $task7.0
cond #1  (1)  set $work0 1; set $work1 0; cont
break/T  vm_fault $task7.0
cond #2  ($work0) set $work1 ($work1+1); cont
break/Tu xxx_end $task7.0
cond #3  ($work0) print $work1 " faults\n"; set $work0 0
cont
```

will print page fault counts from `xxx_start` to `xxx_end` in `task7`.

`step[/p] [,count]`
: Single step *count* times. If `p` option is specified, print each instruction at each step. Otherwise, only print the last instruction.

Warning: depending on machine type, it may not be possible to single-step through some low-level code paths or user space code. On machines with software-emulated single-stepping (e.g., pmax), stepping through code executed by interrupt handlers will probably do the wrong thing.

`continue[/c]`
: Continue execution until a breakpoint or watchpoint. If `/c`, count instructions while executing. Some machines (e.g., pmax) also count loads and stores.

Warning: when counting, the debugger is really silently single-stepping. This means that single-stepping on low-level code may cause strange behavior.

`until`
: Stop at the next call or return instruction.

`next[/p]` Stop at the matching return instruction. If `p` option is specified, print the call nesting depth and the cumulative instruction count at each call or return. Otherwise, only print when the matching return is hit.

`match[/p]`
A synonym for `next`.

`trace[/tu] [ `*`frame_addr|thread`*` ][,`*`count`*`]`
Stack trace. `u` option traces user space; if omitted, only traces kernel space. If `t` option is specified, it shows the stack trace of the specified thread or a default target thread. Otherwise, it shows the stack trace of the current thread from the frame address specified by a parameter or from the current frame. *count* is the number of frames to be traced. If the *count* is omitted, all frames are printed.

Warning: If the target thread's stack is not in the main memory at that time, the stack trace will fail. User space stack trace is valid only if the machine dependent code supports it.

`search[/bhl]` *`addr value`* `[`*`mask`*`] [,`*`count`*`]`
Search memory for a value. This command might fail in interesting ways if it doesn't find the searched-for value. This is because `ddb` doesn't always recover from touching bad memory. The optional count argument limits the search.

`macro` *`name commands`*
Define a debugger macro as *name*. *commands* is a list of commands to be associated with the macro. In the expressions of the command list, a variable `$argxx` can be used to get a parameter passed to the macro. When a macro is called, each argument is evaluated as an expression, and the value is assigned to each parameter, `$arg1`, `$arg2`, . . . respectively. 10 `$arg` variables are reserved to each level of macros, and they can be used as local variables. The nesting of macro can be allowed up to 5 levels. For example,

```
macro xinit set $work0 $arg1
macro xlist examine/m $work0,4; set $work0 *($work0)
xinit *(xxx_list)
xlist
...
```

will print the contents of a list starting from `xxx_list` by each `xlist` command.

`dmacro` *`name`*
Delete the macro named *name*.

`show all threads[/ul]`
Display all tasks and threads information. This version of `ddb` prints more information than previous one. It shows UNIX process information like `ps` for each task. The UNIX process information may not be shown if it is not supported in the machine, or the bottom of the stack of the target task is not in the main memory at that time. It also shows task and thread identification numbers. These numbers can be used to specify a task or a thread symbolically in various commands. The numbers are valid only in the same debugger session. If the execution is resumed again, the numbers may change. The current thread

can be distinguished from others by a # after the thread id instead of :. Without l option, it only shows thread id, thread structure address and the status for each thread. The status consists of 5 letters, R(run), W(wait), S(sus pended), O(swapped out) and N(interruptible), and if corresponding status bit is off, . is printed instead. If l option is specified, more detail information is printed for each thread.

`show task [ addr ]`
: Display the information of a task specified by *addr*. If *addr* is omitted, current task information is displayed.

`show thread [ addr ]`
: Display the information of a thread specified by *addr*. If *addr* is omitted, current thread information is displayed.

`show registers[/tu [ thread ]]`
: Display the register set. Target thread can be specified with t option and *thread* parameter. If u option is specified, it displays user registers instead of kernel or currently saved one.

  Warning: The support of t and u option depends on the machine. If not supported, incorrect information will be displayed.

`show map addr`
: Prints the `vm_map` at *addr*.

`show object addr`
: Prints the `vm_object` at *addr*.

`show page addr`
: Prints the `vm_page` structure at *addr*.

`show port addr`
: Prints the `ipc_port` structure at *addr*.

`show ipc_port[/t [ thread ]]`
: Prints all `ipc_port` structure's addresses the target thread has. The target thread is a current thread or that specified by a parameter.

`show macro [ name ]`
: Show the definitions of macros. If *name* is specified, only the definition of it is displayed. Otherwise, definitions of all macros are displayed.

`show watches`
: Displays all watchpoints.

`watch[/T] addr,size [ task ]`
: Set a watchpoint for a region. Execution stops when an attempt to modify the region occurs. The *size* argument defaults to 4. Without T option, *addr* is assumed to be a kernel address. If you want to set a watch point in user space, specify T and *task* parameter where the address belongs to. If the *task* parameter is omitted, a task of the default target thread or a current task is assumed. If you specify a wrong space address, the request is rejected with an error message.

Warning: Attempts to watch wired kernel memory may cause unrecoverable error in some systems such as i386. Watchpoints on user addresses work best.

11.3 Variables

The debugger accesses registers and variables as $*name*. Register names are as in the `show registers` command. Some variables are suffixed with numbers, and may have some modifier following a colon immediately after the variable name. For example, register variables can have `u` and `t` modifier to indicate user register and that of a default target thread instead of that of the current thread (e.g. `$eax:tu`).

Built-in variables currently supported are:

`task`*xx*`[.`*yy*`]`
: Task or thread structure address. *xx* and *yy* are task and thread identification numbers printed by a `show all threads` command respectively. This variable is read only.

`thread`
: The default target thread. The value is used when `t` option is specified without explicit thread structure address parameter in command lines or expression evaluation.

`radix`
: Input and output radix

`maxoff`
: Addresses are printed as *symbol+offset* unless offset is greater than maxoff.

`maxwidth`
: The width of the displayed line.

`lines`
: The number of lines. It is used by `more` feature.

`tabstops`
: Tab stop width.

`arg`*xx*
: Parameters passed to a macro. *xx* can be 1 to 10.

`work`*xx*
: Work variable. *xx* can be 0 to 31.

11.4 Expressions

Almost all expression operators in C are supported except `~`, `^`, and unary `&`. Special rules in `ddb` are:

`identifier`
: name of a symbol. It is translated to the address(or value) of it. `.` and `:` can be used in the identifier. If supported by an object format dependent routine, [*file_name*:]*func*[:*line_number*] [*file_name*:]*variable*, and *file_name*[:*line_number*] can be accepted as a symbol. The symbol may be prefixed with *`symbol_table_name`*`::` like `emulator::mach_msg_trap` to specify other than kernel symbols.

`number`
: radix is determined by the first two letters:

  `0x` hex

  `0o` octal

  `0t` decimal

otherwise, follow current radix.

`.` dot

`+` next

`..` address of the start of the last line examined. Unlike dot or next, this is only changed by `examine` or `write` command.

last address explicitly specified.

`$`*`variable`*
: register name or variable. It is translated to the value of it. It may be followed by a : and modifiers as described above.

`a` multiple of right hand side.

`*`*`expr`* indirection. It may be followed by a : and modifiers as described above.

Appendix A GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

A.0.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

A.0.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General

Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.
5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose

any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two

goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Appendix B Documentation License

This manual is copyrighted and licensed under the GNU Free Documentation license.

Parts of this manual are derived from the Mach manual packages originally provided by Carnegie Mellon University.

B.1 GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

   D. Preserve all the copyright notices of the Document.

   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

   F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

   G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

   H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

B.1.0.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

B.2 CMU License

Mach Operating System
Copyright © 1991,1990,1989 Carnegie Mellon University
All Rights Reserved.

Permission to use, copy, modify and distribute this software and its documentation is hereby granted, provided that both the copyright notice and this permission notice appear in all copies of the software, derivative works or modified versions, and any portions thereof, and that both notices appear in supporting documentation.

CARNEGIE MELLON ALLOWS FREE USE OF THIS SOFTWARE IN ITS "AS IS" CONDITION. CARNEGIE MELLON DISCLAIMS ANY LIABILITY OF ANY KIND FOR ANY DAMAGES WHATSOEVER RESULTING FROM THE USE OF THIS SOFTWARE.

Carnegie Mellon requests users of this software to return to

Software Distribution Coordinator
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3890

or `Software.Distribution@CS.CMU.EDU` any improvements or extensions that they make and grant Carnegie Mellon the rights to redistribute these changes.

Concept Index

Function and Data Index

C

D

E

H

I

M

P

S

T

V

Short Contents

Table of Contents