MK7.3
Release Notes

By
The Open Group Research Institute's Real-Time Group

October 30, 1997

THE Open GROUP
RESEARCH
INSTITUTE

11 Cambridge Center
Cambridge, MA 02142

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Materiel Command (AFMC).

Table of Contents

1 Feature Description

1.1 Introduction

This document is an extension and enhancement of the previously issued MK5.0 Release Notes and covers the modifications and enhancements to the MK system from MK5.0 through MK6.3, MK7.1, MK7.2, and MK7.3. The MK7.3 release comprises results from a number of activities and projects at the The Open Group Research Institute (RI).[1] These are:

a. The general evolution of the microkernel interface.

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

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

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

1. The Research Institute was previously known as the Open Software Foundation Research Institute, prior to the merge of the Open Software Foundation and X/Open to form The Open Group.

e. The RT2 real-time project (MK6.1), which has contributed fundamental extensions to the kernel in support of real-time applications and environments.

f. The MicroKernel Performance Suite (MKPS).

g. The Event Trace Analysis Package (ETAP), which is designed to provide users with tools to study system behavior.

h. IP Multicast, which allows the transmission of an IP datagram to a “host group”, a set of zero or more hosts identified by a single IP multicast destination address.

i. Additional device drivers for the parallel port, DEC DE-450 Etherworks PCI card, the Myrinet PCI board, PCI SCSI adapters, DEC FDDI PCI board and the SoundBlaster ISA audio card.

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

1.2 New and Enhanced Features in MK7.3

MK7.3 contains a number of specific new features, as well as enhancements of already existing features, including:

a. CORDS project enhancements, which, among other things, consolidate previously released capabilities.

b. Clock and timer extensions and improvements, increasing the stability of the NTP distributed clock, as well as providing faster access to a more accurate and precise system clock.

c. Debugger enhancements for multi-threaded programs.

d. Access to a wider variety of real-time services through POSIX specified interfaces[30], including a substantial portion of the POSIX thread and real-time features.

e. Initial implementation of the MK Scheduling Framework and a set of scheduling policies, including POSIX-defined policies and an experimental, Alpha OS-derived best-effort processor scheduling policy.

f. An experimental, microkernel-provided, trans-node RPC facility.

g. Myrinet is now supported, both as a local network interconnect and as an integrated part of the CORDS subsystem.

h. Restructuring and clean-up in several areas: thread control, preemption and locking, and the stack-per-shuttle optimization.

i. Inclusion of a number of test programs, including an interrupt latency benchmark.

These features are described in the following sections.

1.2.1 CORDS Project Enhancements

The MK7.3 release consolidates the CORDS technologies that were first offered with MK7.2. These technologies are GIPC (Group IPC) and its companion set of applications, which exercise group communication, fault isolation, and fault management properties.

GIPC's inner protocol constituents have been substantially hardened. For instance, protocols involved in maintaining node liveness have been enhanced with “skeptics”; skeptics recognize and mask status changes that occur too often within a given interval, due to failure modes (e.g., hardware flickers) or malicious behaviors. The membership protocol has also been improved to more quickly converge to a stable membership view, even when some of the nodes are not responsive and need to be evicted from a proposed membership view. The lower layers of GIPC protocol graphs have been adapted to support multiple interconnects (e.g., Ethernet and Myrinet); while two or more interconnects are still mutually exclusive, we anticipate taking full advantage of redundant interconnects for superior failure resilience. GIPC's API has not changed from MK7.2.

The two demonstration applications delivered with GIPC have also been revamped to improve their efficacy. The marker demonstration (see the man page for *demo_client*(8)) is now a complete outline of a distributed application whose constituents communicate using data sets that depend upon the agreed actual population of constituents; the demonstration proves the effectiveness of GIPC's virtual synchrony model, the viability of GIPC API and locking within a multi-threaded GIPC client. The ball-handler demonstration (see the man page for *balldemo*(8)), which requires a special mechanical apparatus, takes advantage of the experience achieved in offering the demonstration at several locations outside of the RI. For more information on the ball-handler demonstration, see:

http://www.opengroup.org/RI/ri/AdvisBrd/Oct96_Mtg/present/ft-demo/ftdemo.htm

The CORDS framework has been restructured to use either of two thread packages: *cthreads* or POSIX threads (*pthreads*). While *cthreads* is still the default choice of thread model, we anticipate shifting our focus to *pthreads*, due to its real-time viability. (Conversely, *cthreads* circumvents kernel scheduling decisions.) For further information on the use of *pthreads* in CORDS, please contact Franco Travostino (f.travostino@opengroup.org).

In-kernel CORDS configurations compiled with XK_ETAP can now be traced with ETAP. (There is a unique ETAP event for all CORDS activities). As part of the 'nosupport' deliverables, we introduce our experimental toolkit for off-line reconciliation of ETAP trace logs from different nodes (Distributed ETAP, or DETAP). The enabling technology for DETAP is CORDS' distributed clock synchronization facility (LTS).

1.2.2 Clock and Timer Extensions and Improvements

MK7.3 provides a number of improvements in the area of clocks and timers, including increased stability of the NTP[19] distributed clock, as well as faster access to a more accurate system clock. These improvements are documented in [20].

One objective of this work was to improve the clock OS support code to deal with the rather loose crystal accuracy characteristics evident in typical machines.

Another objective was to investigate and resolve problems that some users of the MK system had observed. These included: (1) reported time intervals that were impossibly small or negative; (2) inadequate NTP synchronization; and (3) excessive overhead to read the clock.

This last area, providing low-overhead, high-resolution clock reading, was particularly interesting given the fact that newer microprocessors, including the Pentium, have some form of "cycle counter" — i.e., a large register that counts at a high rate — that looks like a very good candidate for a timekeeping device. And, on the Pentium at least, this register can be made readable in any mode (including user) and so seems to offer the prospect of a clock that can be read with very low overhead.

MK7.3 offers the results of this work. Among the highlights:

a. The primary cause of anomalous time events (negative intervals) was a logic flaw in the 8254 Programmable Interval Timer implementation on one particular hardware platform. MK7.3 includes a software workaround.

b. The assumed frequency of the 8254 clock source is now a dynamically adjustable attribute.

c. A long-term feedback loop utilizing NTP tunes the 8254 frequency attribute to within 1 part-per-million.

d. The UNIX server now separates and modularizes UTC (Coordinated Universal Time) variables from those variables involved in local real-time timekeeping. Moreover, UTC-awareness has been removed from the kernel, and the execution time of *gettimeofday()* has been reduced by almost 50%. The execution time of the POSIX-compliant function *clock_gettime()* has been reduced by about 99%, and this is the recommended method for all applications to get the current time of day.

e. The Pentium Time Stamp Counter is now utilized as a very high precision, very low overhead, real-time clock device offering the same long-term accuracy. The result is a system clock with precision, speed, and accuracy heretofore only available with special purpose hardware.

As is described elsewhere in this chapter, all of this work complies to the current POSIX standard for clocks and timers.

1.2.3 Debugger Enhancements for Multi-Threaded Programs

Earlier MK7 releases provided separate versions of the 'gdb' debugger for use with single-threaded programs (based on *ptrace*) and for multi-threaded programs (where 'gdb' must be MK aware). For MK7.3, these functions have been merged so that there is now a single version of 'gdb,' which is both *ptrace* and MK-aware, and which can be found in:

- /usr/bin/gdb

It is based on the Free Software Foundation's 'gdb' version 4.16 and works with executables compiled by 'gcc' version 2.7.1, which is the version of 'gcc' that has been used with MK7.2 and MK7.3.

Conceptually, this version of 'gdb' executes in two distinct modes. By default it executes as *ptrace* 'gdb,' and it can be used in the traditional manner (without requiring special privileges or additional support) to debug single-threaded programs.

However, to use this version of 'gdb' to debug multi-threaded programs, the user does need additional privilege and support. In particular, the user must be 'root' and the 'machid' program must be running. Then, after starting 'gdb,' the user should issue the following command:

- (gdb) target mach

At this point, for executables linked with 'libmach,' the user can issue the 'run' command to start the program. For those executables that have not been linked with ‘libmach,’ the ‘run’ command should not be used. Instead, ‘attach’ should be used. (Multi-threaded executables will probably have been linked with ‘libmach.’)

You can use the standard ‘gdb’ commands. Additional commands specific to multi-threading include:

- “tl” — for thread list;
- “ts” — for thread select;
- “thread-tree” — to print threads as a stack tree;
- “help mthread” — to list all of the multi-thread subcommands.

1.2.4 POSIX-Compliant Access to Real-Time Services

The RI is working to advance the state-of-the-art of scalable, embedded real-time applications by developing an operating system for scalable embedded real-time systems that complies to the real-time POSIX standard[30] and that builds upon the proven MK Microkernel, which provides scalability, performance, real-time, and instrumentation capabilities.

The POSIX series of standards specifies application and user interfaces to open systems. They define the application interfaces to basic system services for input/output, file system access, and process management. They also include interfaces appropriate for real-time applications and interfaces appropriate for multi-threaded applications. (All of the POSIX standards cited here correspond to interfaces expressed in the C programming language.)

We have identified a functional subset of these POSIX interfaces that will provide a small, simple, and *fast* standards-subset OS environment specifically targeted at scalable, embedded real-time applications. The POSIX real-time services will include most of the thread, memory management, clocks and timers, and synchronization subsystems.

We are providing POSIX services in two phases. The first phase provides a POSIX threads run-time library, which is included in MK7.3 and is described below.

The second phase, which will be provided in future operating system releases, will produce a more full-featured POSIX facility, utilizing one or more servers operating on top of the MK Microkernel. The resulting system will support distributed, fault-tolerant, real-time applications, employing MK Microkernel facilities to provide POSIX-compliant services in the most advantageous manner.

POSIX standards define a number of conforming implementation options, which correspond to groups of POSIX routines — for example, _POSIX_PRIORITY_SCHEDULING or _POSIX_REALTIME_SIGNALS. These implementation options allow a given POSIX implementation to maintain POSIX conformance while providing only a subset of the POSIX-specified routines (as opposed to *all* POSIX-specified routines). Presumably, a subset is provided to satisfy the requirements of some intended audience.

In MK7.3, none of these POSIX conformance options have been set because we do not conform to the POSIX specification at this time. However, for those routines that we have implemented, the interfaces comply to the POSIX specification, as do the resultant behaviors — to the extent that is currently reasonable.

1.2.4.1 The POSIX Threads Run-Time Library

The POSIX standard[30] specifies support routines for multi-threaded applications in a real-time environment. We have implemented a considerable subset of these routines, and have chosen to implement them directly on top of the MK Microkernel for two reasons. First, the POSIX thread model is very similar in nature to the native thread model found in the MK system. Secondly, having this close relationship between POSIX threads and MK Microkernel threads allows us to exploit fully the technologies we are creating for real-time systems. In this way, the POSIX thread library can take advantage of such things as migrating threads and the real-time scheduling framework.

Currently supported are:

- threads
- thread attributes (with some limitations)
- mutex variables
- mutex variable attributes
- condition variables
- “once” variables
- thread-specific data

Some POSIX-specified features have been left out so that we can support POSIX-style thread programming in the absence of a complete POSIX (server) environment. Features that require server support will be implemented as the POSIX Server becomes available in future system releases.

For this first implementation of the POSIX Threads Run-Time Library, we do not support:

a. thread-level signals
   Since the MK real-time system philosophy is based on cooperating threads using RPC and shared memory for communication, signals are not needed; and, in fact, they conflict with the provision of end-to-end real-time performance and time management.

b. sharing of thread data structures between separate tasks
   The POSIX standard allows for mutex and condition variables to be shared between separate tasks. We don't support this in order to provide a more efficient implementation of these synchronizers.

c. "heavyweight" POSIX *fork()* support
   Currently we support a single task, multiple thread application model. We will add *fork()* support if a multiple task model requires it.

d. thread stack attributes
   The POSIX standard supports thread attributes which control the size and placement of the run-time stack for the thread. In this implementation, we provide for a single system wide control over the size (granularity) of thread stacks, but not their placement. This is primarily to allow for a more efficient implementation of thread-specific data structures used by the library itself.

e. making the POSIX run-time library "thread-safe" and "thread-friendly"
   We intend to provide a fully thread-safe and thread-friendly POSIX run-time library in the future. This library will continue to comprise a standards-compliant subset of the full POSIX specification [30].

1.2.4.2 Additional Notes Concerning the POSIX Threads Run-Time (*pthreads*) Library

a. There are actually two *pthreads* libraries available in MK7.3. The new *pthreads* library described above has header and library files in the "standard" places, specifically:
   - /usr/include/pthread.h
   - /usr/ccs/lib/libpthread.a

   In addition, there is an older *pthreads* library, corresponding to Draft 4 of the POSIX System Application Program Interface, which constitutes part of the OSF/1 API that was added to support DCE (Distributed Computing Environment). The older *pthreads* library is based on OSF/1 system calls rather than MK microkernel primitives. This has certain consequences, notably that it is available to unprivileged programs, where microkernel system calls are not.

The corresponding header and library files for the older version of the *pthreads* library are found in:

- /usr/include/posix_draft_4/exc_handling.h
- /usr/include/posix_draft_4/pthread.h
- /usr/include/posix_draft_4/pthread_exc.h
- /usr/ccs/lib/posix_draft_4/libpthreads.a

b. The new *pthread* routines comply to POSIX specifications, with these routines currently unimplemented:
pthread_atfork()
pthread_attr_getscope()
pthread_attr_setscope()
pthread_attr_getstackaddr()
pthread_attr_getstacksize()
pthread_attr_setstackaddr()
pthread_attr_setstacksize()
pthread_condattr_getpshared()
pthread_condaddr_setpshared()
pthread_kill()
pthread_sigmask()
pthread_mutexattr_getpshared()
pthread_mutexattr_setpshared()

c. The new *pthreads* library does not currently offer 32 priority levels for real-time system configurations. Rather, in MK7.3, 30 priority levels are available for user threads. This deviation from the POSIX specification will be addressed in future releases.

d. Access to the Best-Effort scheduling policy has not yet been integrated into the *pthreads* library routines. For the time being, applications should use the microkernel-level '*thread_set_sched()*' system call to access the Best-Effort scheduling policy. Best-Effort support will be integrated into the *pthreads* library in a future release.

e. Do not use '*mach_thread_self()*' to get the kernel port for a *pthread* because this will generate a thread reference that the *pthread* library does not know about and can result in resource leakage. Rather, call *_pthread_kernel_port(pthread_self())*.

f. If possible, MK users should use the *pthreads* library, rather than either *cthreads* or *rthreads*, since this is the library on which future work will focus. Previously, real-time users were advised to use *rthreads*. Many of those users should be able to transition to *pthreads* now, but some cannot due, for instance, to the fact that *rthreads* is currently more integrated with the migrating thread abstraction than is *pthreads*.

1.2.4.3 Clocks and Timers

MK7.3 features additional and improved clock and timer functionality relative to previous releases, providing users with higher clock resolution, lower overhead to access the clock, and more accurate time. (This work is described in more detail elsewhere in this chapter.)

As of this release, all implemented clock and timer functions comply with the POSIX specification. Not all of the specified functions in the clocks and timers chapter are implemented, so _POSIX_TIMERS is not set. In particular, the following clock and timer functions are currently unimplemented:

clock_getres()
nanosleep()

1.2.5 MK Scheduling Framework and Scheduling Policies

MK7.3 includes the initial implementation of the MK Scheduling Framework and a set of scheduling policies, which comprises not only POSIX-specified processor scheduling policies, but also a time-driven Best-Effort scheduling policy derived from work done on the Alpha Operating System, which performed research into the support of distributed, fault tolerant, real-time applications[14][9].

MK7.3, including the new Scheduling Framework, is as robust as its predecessors with respect to scheduling. That is, the introduction of the Scheduling Framework and its attendant system restructuring has not decreased system robustness based on the RI's testing to date.

Nonetheless, this work is ongoing, and it is anticipated that the Scheduling Framework interfaces will continue to evolve as more scheduling policies are implemented and as richer scheduling capabilities are pursued. Similarly, the scheduling policy implementations might evolve in the future, particularly those for new policies such as Best-Effort. Despite this evolution, though, the existing POSIX-defined scheduling policies are expected to remain stable and are not expected to display any user-visible changes.

The MK Scheduling Framework and its policies have not been tested on SMP platforms.

1.2.5.1 The Scheduling Framework

The MK scheduling subsystem, which provides the policies and mechanisms to schedule and control threads, has been restructured in MK7.3 to incorporate the MK Scheduling Framework to:

a. separate scheduling policies from scheduling mechanisms,

b. support multiple independent scheduling policies concurrently, and

c. enable new functionality for real-time support.

The Framework provides access to scheduling policy interfaces for the kernel and provides kernel mechanisms for use by scheduling policies. These mechanisms, which include, for example, context switching, run queue manipulation, and timer facilities, do not change when different scheduling policies are installed. These mechanisms can themselves be replaced but they are considered policy independent.

The Framework design is based on the philosophy that each policy should be treated as a black box. A policy exports its interfaces to the Scheduling Framework which, in turn, provides access to the policy for the rest of the kernel and the other policies.

To mediate interactions among policies, each policy associates an attribute that is universally understood within the framework with each thread it schedules. This attribute is a *metapriority*, an integer value from a bounded range that gives the kernel a low-overhead mechanism for evaluating the relative importance of two threads, even those scheduled by different policies.

The Framework always executes the ready thread with the best metapriority. Whenever a ready thread's metapriority is better than that of the currently running thread, the Framework preempts the current thread.

Policies can have overlapping metapriority ranges, although no metapriority can be shared by two or more policies. The purpose of a policy's metapriority range is to allow the relative importance of a policy's threads to be easily configurable. Allowing overlapping ranges precludes a policy from implicitly locking out the threads of another policy.

Each scheduling policy is represented by a *policy object*, which defines the interfaces (including a small amount of shared data) between the policy and the framework. The framework, in turn, provides access to these interfaces for the rest of the kernel.

Each policy object is characterized by the following policy-specific data:

- policy ID— A unique policy number.
- priority mask — An explicit indication of the metapriorities managed by the policy.
- scheduling attributes size — The size of the scheduling parameters data structure for this scheduling policy. These are the parameters that are actually communicated — and meaningful — outside the policy. (This is as opposed to the internal state, which has some overlap with the scheduling parameters.)
- scheduling information size — The size of the scheduling policy's internal state that describes the state of an individual thread.
- standard scheduling operations — The set of policy-specific operations that correspond to the general interfaces for policies exported by the MK Scheduling Framework.

The standard scheduling operations provided by a scheduling policy include:

- *sp_thread_attach()* — Associate a thread with a scheduling policy.
- *sp_thread_detach()* — Disassociate a thread from a scheduling policy.
- *sp_thread_begin()* — Notify the policy that a new thread has been selected to run
- *sp_thread_done()* — Notify the policy that the current thread is done
- *sp_thread_dispatch()* — Notify the policy that an old thread is ready to be requeued.
- *sp_thread_unblock()* — Notify the policy that a thread has become runnable
- *sp_thread_set()* — Set a thread's scheduling attributes.
- *sp_thread_get()* — Get a thread's scheduling attributes.
- *sp_alarm_expired()* — Handle an expired alarm condition.

1.2.5.2 Scheduling Policies

The Timesharing, FIFO, and Round Robin scheduling policies that were previously supported on MK systems continue to be supported in MK7.3 systems, and are referred to as the *MK Scheduling Policies*. Kernel and UNIX server threads utilize the MK Scheduling Policies.

The FIFO and Round Robin policies correspond to the 'SCHED_FIFO' and 'SCHED_RR' policies described in the POSIX standard[30].

In addition to the MK Scheduling Policies, MK7.3 contains a time-driven scheduling policy called the Best-Effort Scheduling Policy. This policy is based on previous implementations in the Alpha Operating System[16][21]. It is intended to be useful in a dynamic real-time system, where processing overloads can occur. It attempts to provide a mechanism for graceful degradation during periods of overload.

The Best-Effort Scheduling Policy bases its scheduling decisions on time constraints that are explicitly declared by an application. The application also supplies estimates of the expected execution time needed for each time-constrained computation. In the event that there are sufficient processing cycles to satisfy all specified time constraints, then all threads are scheduled in "deadline order." However, if a processor is overloaded, then threads are shed based on a heuristic that attempts to maximize the value accrued by the application (as specified by the time constraints in application-specific units).

The following items should be considered when using multiple policies in the framework:

a. The MK Scheduling Policies, which are needed to run the UNIX Server, are always present.

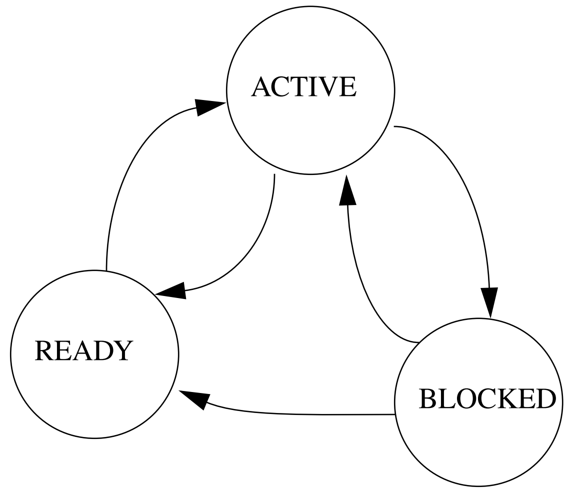
b. Only one real-time policy — that is, a policy that bases scheduling decisions on explicitly specified application time constraints, such as Earliest Deadline First or Best-Effort — should be active. (Note that, by this definition, none of the MK Scheduling Policies are real-time policies.)

1.2.5.3 Scheduler State Machine

The original Mach 3 scheduling state diagram was large and complex. The separation of scheduling mechanisms from scheduling policies implemented in MK7.3 partitions some of the scheduling state machine into the scheduling mechanisms, which are policy independent. Consequently, policy designers need only concern themselves with that portion of the state machine that varies from policy to policy.

Conceptually, the Scheduling Framework provides a scheduling policy with a simplified scheduling state machine in which threads can be in one of only three states:

- ACTIVE — running on a processor. *thread_dispatch()* moves the thread into BLOCKED if WAIT is set; otherwise, it moves it into READY.
- READY — runnable. *thread_invoke()* moves the thread into ACTIVE.
- BLOCKED — not runnable. *thread_setrun()* moves the thread into ACTIVE if there is an idle processor; otherwise it moves it into READY.



Thread State Transitions

1.2.6 Experimental Trans-Node RPC

MK7.3 has introduced support for Remote Procedure Calls (RPCs) that cross node boundaries. This mechanism is based on the notion of "migrating threads," whereby scheduling (and other) information associated with the client thread flows along with any data to the server[4].

The server thread, possibly on another node, will then assume the scheduling characteristics of the client thread for the duration of the RPC function.

The current version uses KKT (Kernel-to-Kernel Transport) as the transport between the two nodes. Also currently required is a "DIPC" configuration for support of multi-node systems and communication paths between them.

1.2.7 Myrinet Support

MK7.3 introduces support for Myrinet. Like Ethernet and FDDI, Myrinet has been integrated with CORDS and can be used as an interconnect for CORDS protocols. In addition, Myrinet can also be used without CORDS as an another computer interconnect.

Information about Myrinet is available from Myricom at:

http://www.myri.com

1.2.8 Restructuring and Clean-Up

Although these changes should be generally invisible to users, several areas — including thread control, preemption and locking, and the association of a single stack per shuttle — have been made in MK7.3.

The thread control operations have been redesigned to work more reliably, to simplify the scheduling state machine, and to better support RPC. In order for thread control operations to behave reliably, they only effect the user visible state of a thread. A thread must be in user mode or blocked in a "user visible" condition before thread control operations can be applied.

Changes in thread attributes (e.g. registers, scheduling attributes) will be processed immediately if the thread is in the correct state. Otherwise, the requester will block until the thread reaches an acceptable state. Changes in thread state (e.g., *thread_suspend()*, *thread_abort()* and alerts along an RPC chain) will also block until the thread is in the appropriate state.

To improve both space-efficiency and performance, there is now a single kernel stack associated with each shuttle — that is, the schedulable portion of a migrating thread. Previously, there had been a kernel stack associated with each activation comprising a migrating thread.

1.2.9 Test Programs

Several test programs have been included in MK7.3. Typically, these test basic system features, such as kernel synchronizer performance or the correctness and performance of the *pthreads* library functions. There is also a test that measures the interrupt latency for the system.

1.3 Collocated Servers

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

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

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

More information can be found in a Research Institute paper [10].

1.4 Asynchronous Driver Model

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

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

1.5 Real-Time Extensions

This release is a continuation of the effort by the RI to provide support for real-time applications and environments. The major areas of real-time functionality provided in MK7.3 are described in this section.

1.5.1 Preemptible Kernel

One of the projects undertaken by the RI as part of the kernel interface unification effort is the design and implementation of several changes and enhancements to MK to improve event response and context switch latency. These modifications include the following[1]:

- Implementation of high-performance blocking locks to replace most simple spinlocks.
- Elimination of all complex blocking locks, which can cause arbitrarily long scheduler invocations due to the "thundering herd" wakeup effect. (MK, like UNIX before it, has a feature wherein an arbitrary number of processes/threads can wait for a particular event to occur; when it does, they are all awakened simultaneously; hence, the thundering herd.)

1.5.2 Synchronizers

MK7.3 supports two types of synchronizers: semaphores and mutual-exclusion locks (see [5] for more information).

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

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

1.5.3 Real-Time Threads (*rthreads* and *pthreads*)

The RT Threads Library is a user-level library and is not, therefore, documented in the Kernel API Changes chapter of the Release Notes. See [5] for a complete description of the library functionality.

---

1. Currently, kernel preemption only works on uniprocessor systems.

The POSIX threads (*pthreads*) library, not the *rthreads* library, is the focus of current and anticipated future development by the RI in this area. Consequently, the RI advises users to switch to *pthreads* where appropriate and discourages the use of *rthreads* when it can be avoided.

1.5.4 Extended Real-Time Thread Control Semantics

The thread_suspend function blocks the targeted thread to stabilize its user visible state. The successful return from thread_suspend guarantees the caller that the targeted thread is no longer executing user code.

If a suspended thread is involved in a system call that changes its user visible state, the state retrieved will always be consistent. Under these circumstances, if the suspended thread's state is changed by another thread, it is intended that the controlling thread's changes should supersede those of the suspended thread.

In the circumstances above, if a controlling thread suspends another thread, retrieves user visible state of the suspended thread, modifies the state, and resumes the thread, either of the following is possible:

- the suspended thread's changes to itself haven't taken effect:
  the retrieve returns the previous values; the modify updates the values; after the resume, the target will return from its system call indicating that the call was aborted
- the suspended thread's changes to itself have taken effect:
  the retrieve returns the values it set; the modify updates the values; after the resume, the target will return from its system call indicating success

In either case, the modifications specified by the controlling thread should be effective after the suspended thread is resumed.

1.5.5 RPC Kernel Interface

Recent work on existing systems indicates that message based systems that optimize the Remote Procedure Call (RPC) case experience significant performance improvements. This work leverages the collocation effort and the redefinition of a thread into two components: a thread and an activation.

Perhaps the most important difference between the RPC mechanism and Mach IPC based RPC is the treatment of message meta data. The RPC parameter data, and the IN and OUT arguments to the remote service, are treated separately from the meta data. Every effort will be made to use the parameter data, without additional copies, as it is provided by the standard ‘C’ compiler of each target platform. This contrasts with the **mach_msg** approach which requires the parameter data to be copied to the message body. It is possible to send either a message or an RPC to the same port. For a more detailed description of this functionality, see the included paper [4].

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

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

1.6 CORDS

CORDS (Communication Objects for Real-time Dependable Systems) is a configurable, object-oriented Communication Subsystem derived from the University of Arizona’s x-kernel. CORDS has a number of desirable real-time features: virtual channels, fine control over system resources, privileged messaging and queue-free code paths. CORDS is execution domain independent: a protocol stack, generalized to a protocol graph, can be extended across one or more protection domain boundaries (i.e., the kernel/user boundary) without changes to individual protocols. CORDS’ dependencies on the host OS are all resolved within a limited portion of the CORDS framework; thus, it is conceivable to rehost CORDS to different OSs.

Many protocols have been implemented and are available in CORDS, including the most popular Internet protocols (i.e., TCP, UDP, IP, BOOTP, etc.), non-standard protocols developed at University of Arizona (i.e., CHAN, BLAST, etc.), and protocols developed at the RI for distributed systems (i.e., KKT, reliable and ordered group communication, and time synchronization protocols).

In MK7.3, the above mentioned protocols and/or new, user-provided protocols can be combined to build arbitrary CORDS protocol graphs.A set of built-in MK7.3 kernel configurations — namely {DEBUG, FAST, PRODUCTION}+NORMA+net — already include a CORDS protocol graph which complements DIPC with the communication fabric required by Ethernet-based NORMA clusters.

Two additional protocols were added to the CORDS environment in MK7.2 to support the use of Digital Equipment Corporation's DEFPA FDDI module for the PCI bus:

- fddi

  This is a hardware independent protocol that provides an interface between the actual FDDI drivers and the rest of CORDS. There should be a separate instantiation of the FDDI protocol for each driver protocol.

- xkfta

  MK in-kernel driver for the DEFPA module with microkernel and CORDS interfaces. Each instantiation is associated with a single FDDI device. XKFTA requires no lower protocol. The last digit of the instance name must correspond to the unit device number for a particular device. For example:

  name=xkfta/fta0

  for device 0.

1.6.1 CORDS Evolution

CORDS has grown along both the framework and protocol dimensions. The key evidence of such a growth can be found in the successful demonstration of Real-Time, Fault Tolerant behaviors, as presented in public events. One such demonstration is documented in:

http://www.opengroup.org/RI/OS/CORDS/ftsorter.htm

Additional information about CORDS can be found in:

http://www.opengroup.org/RI/PubProjPgs/CORDS.htm

The framework has been enhanced with additional real-time features: new tunable parameters have been exposed, for the user to operate more "knobs" such as scheduling attributes of threads and event granularity. The Path[28] interface[18] has been extended as well. In kernel space, the framework can now operate safely in the preemptive environment provided with the real-time enabled kernel configurations. In user space, the framework can optionally be built to use *rthreads*[5] and synchronizers[6] rather than traditional *cthreads*. Moreover, the framework can also be built to use the *pthreads* library. Although the *pthreads*-based configuration is not officially supported in MK 7.3, it will become the preferred configuration in the future. Please note that the choice of thread paradigm has no impact on protocol code.

In the realm of protocols, the original protocols were significantly enhanced and consolidated for the MK7.2 distribution. These protocols implement clock synchronization, IP multicast, node-liveness, and ordered group communication. In addition to Ethernet, CORDS protocols can now access an FDDI and a Myrinet network interface, both from kernel space and user space.

Within the CORDS space, two protocol stacks were implemented as servers in MK7.2: the LTS and GIPC servers.

1.6.2 LTS

LTS[12] provides a distributed clock with a guaranteed upper bound on the difference between the master clock and any of its client clocks. If the service becomes unable to maintain this guarantee, the client clock explicitly fails. These properties differentiate LTS from more traditional approaches to clock synchronization[19]; they make LTS an important building block of real-time distributed systems.

The LTS service is highly configurable. The server consists of a user-space management protocol and in-kernel protocols which perform the synchronization and maintain an MK kernel clock object using this data. Applications access the distributed time maintained by LTS using the standard MK clock object interface.

The synchronization and clock control algorithms are based on the work of F. Cristian et. al.[8].

1.6.3 GIPC

Our quest for predictable behaviors in fault isolation, fault management, and group communication[29] produced the first implementation in MK7.2: the Group IPC (GIPC) Developer’s Kit. The kit includes several GIPC server configurations, a client library and a set of clients: tests and visual demonstrations. These clients are already instrumented to validate real-time properties of group interaction.

The GIPC server implements a single type of membership (live but not accurate), and a single type of ordered group communication and delivery semantics (FIFO atomic broadcast with safe delivery). In addition, the GIPC server exports a minimal process group interface, with one process client per physical node and only one group supported.

With the proper configuration of protocols and tunable parameters, the MK7.3 GIPC server Developer's Kit detects node failures and establishes a new, stable membership view in less than 100 milliseconds.[1]

1.7 Microkernel Performance Suite

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

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

1.8 Event Trace Analysis Package

The Event Trace Analysis Package (ETAP) is an MK tracing facility, designed to provide users with a robust set of kernel level facilities to study system behavior. ETAP supports both kernel internal and application level event tracing. Kernel event tracing provides system programmers with a powerful tool for monitoring kernel internal activity. Application level trace support provides application programmers with a consistent, native to the microkernel, model for application specific event tracing. Together, the two levels of tracing enable users to better understand the interaction between their applications and the MK operating system. (See [7].)

1. on a private, isolated 10Mbit/second Ethernet.

1.9 IP Multicast[1]

In the MK7.2 release, a new IP Multicast feature was added with full compliance with the existing standards. For a more complete description of this standard protocol see [11].

IP multicasting is the transmission of an IP datagram to a "host group", a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same "best-efforts" reliability as regular unicast IP datagrams, i.e., the datagram is not guaranteed to arrive intact at all members of the destination group or in the same order relative to other datagrams.

The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time. A host need not be a member of a group to send datagrams to it.

A host group may be permanent or transient. A permanent group has a well-known, administratively assigned IP address. It is the address, not the membership of the group, that is permanent; at any time a permanent group may have any number of members, even zero. Those IP multicast

---

1. Copyright © 8.2 (Berkeley) 3/21/94

addresses that are not reserved for permanent groups are available for dynamic assignment to transient groups which exist only as long as they have members.

In general, a host cannot assume that datagrams sent to any host group address will reach only the intended hosts, or that datagrams received as a member of a transient host group are intended for the recipient. Misdelivery must be detected at a level above IP, using higher-level identifiers or authentication tokens. Information transmitted to a host group address should be encrypted or governed by administrative routing controls if the sender is concerned about unwanted listeners.

1.10 New Kernel Device Drivers

MK7.3 includes support for the NCR/Symbios PCI adapter 53C810; the DEC DEFPA-AA PCI FDDI board; the DEC DE-450 Etherworks 10 PCI card; the Diamond Stealth PCI 64-bit graphics card; the Myrinet PCI network board, SoundBlaster compatible audio cards, and a data acquisition ADC/DAC interface board.

Support for the DDI 8900 series relay cards has been added to MK. Set status and get status routines exist for setting or checking the state of individual relays. The “cio” device must be in the configuration to make use of this driver.

A device driver for the Quatech DAQ-12 12-bit data acquisition card has been added to MK. Set status and get status routines are provided for controlling the A/D and D/A converters, along with other hardware on the card. WARNING: Portions of this driver have not been fully debugged, use at your own risk!

Note: The DDI 8900 and the Quatech DAQ-12 are both used to support the ball-sorter demonstration (see the man page for *balldemo*(8)).

1.11 Reference Papers

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

[2] Black, David, Randy Smith, “Fast Lightweight IPC: Architecture and Design”, OSF Research Institute, Cambridge, MA, May 1995.

[3] Bryant, Bill, “Design of AD 2, a Distributed UNIX Operating System”, draft document, OSF Research Institute, Cambridge, MA, November 1995.

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

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

[6] CaraDonna, Joseph, Robert Haydt, Franklin Reynolds, “New Synchronization Services for OSF Mach: Synchronizers”, OS Collected Papers vol. 3, OSF Research Institute, Cambridge, MA, April 12, 1994.

[7] Caradonna, Joseph, “The Event Trace Analysis Package: ETAP Users Guide”, OSF Research Institute, Cambridge, MA, February, 1995.

[8] Christian, Flavio, “Probabilistic Clock Synchronization”, Distributed Computing 3:146-158, 1989.

[9] Clark, R.K., E.D. Jensen, and F.D. Reynolds, “An Architectural Overview of the Alpha Real-Time Distributed Kernel,” *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pp. 127-146, Seattle, WA, April, 1992.

[10] Condict, Michael, Don Bolinger, Éamonn McManus, David Mitchell, Steve Lewontin, “Microkernel Modularity With Integrated Kernel Performance”, OSF Research Institute, Cambridge, MA, April, 1994.

[11] Deering, S. “Host extensions for IP multicasting”. 1989 August; Internet RFC 1112, 17 p. (Obsoletes RFC 988, RFC 1054)

[12] Feeney, Laura, “LTS: Algorithms and Implementation”, OSF Research Institute Technical Report, in preparation.

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

[14] Jensen, E.D. and J.D. Northcutt, “Alpha: An Open Operating System for Mission-Critical Real-Time Distributed Systems—An Overview,” *Proceedings of the 1989 Workshop on Operating Systems for Mission-Critical Computing,* ACM Press, 1990.

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

[16] Locke, C.D., “Best-Effort Decision Making for Real-Time Scheduling,” Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, May, 1986.

[17] LoVerso, John, Paul Roy, “The Network Architecture of OSF/1 AD Version 2”, OSF Research Institute, Cambridge, MA, February, 1994.

[18] Menze, Edwin, Franco Travostino, ‘‘The x-kernel-based Communication Subsystem for OSF Mach: Changes to the x-kernel 3.2 API”, Version 1.0, May 1995, OSF Technical Report. (Also available as: *ftp://riftp.osf.org/pub/real_time_papers/xk_prog_man.ps*) .

[19] Mills, David, “Network Time Protocol (Version 3) Specification, Implementation and Analysis.” Network Working Group RFC 1305. March, 1992.

[20] Murphy, Daniel, “Timekeeping Enhancements in OSF/MK”, The Open Group Research Institute, Cambridge, MA, June, 1997.

[21] Northcutt, J.D., R.K. Clark, “The Alpha Operating System: Programming Model,” Alpha Project Technical Report 88021, February 1988.

[22] Noveck, David, “Interposition Mechanism for the OSF/1 AD Version 2 File System”, OSF Research Institute, Cambridge, MA, July, 1994.

[23] Noveck, David, “Token Module for OSF/1 AD Version 2”, OSF Research Institute, Cambridge, MA, July, 1994.

[24] Patience, Simon, Faramarz Rabii, “The Design of the Process Management Component of OSF/1 AD Version 2”, OSF Research Institute, April, 1994.

[25] Rabii, Faramarz, Simon Patience, “The Process Management Architecture of AD Version 2”, OSF Research Institute, Cambridge, MA, October, 1993.

[26] Roy, Paul, David Noveck, Durriya Netterwala, “The File System Architecture of OSF/1 AD Version 2”, OSF Research Institute, Cambridge, MA, August, 1993.

[27] Sears, Steve, David Black, Randall Dean, Michelle Dominijanni, Alan Langerman, “Kernel to Kernel Transport Interface for the Mach Kernel”, OSF Research Institute, Cambridge, MA, April, 1994.

[28] Travostino, Franco, Ed Menze and Franklin Reynolds, ‘‘Paths: Programming with System Resources in Support of Real-Time Distributed Applications”, Proceedings of the 1996 IEEE Workshop on Object-Oriented Real-Time Dependable Systems, Laguna Beach, CA, February 1996.

[29] Travostino, Franco, “Elementary Groupware For Fault Management”, OS Collected Papers vol. 4, OSF Research Institute, Cambridge, MA, October 1995.

[30] —, “ISO/IEC 9945-1: Second Edition 1996-07-12 (ANSI/IEEE Std 1003.1), Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language],” Institute of Electrical and Electronics Engineers, New York, NY, 1996.

2 Installation Notes

2.1 Introduction

This chapter describes the procedures for installing and configuring the MK7.3 binary distribution and compiling the source code on the supported reference platform. The distribution tape only includes the source code needed to build the MK7.3 microkernel and OSF/1 server, together with supporting MK-related software. This software must be built on a machine that is already running MK7.1 or later. Once you have installed the binary portion of the MK7.3 release, you can then use the same machine to compile the sources.

Updated information about the release can be found on the World Wide Web by following the "MK7.3 Release Additional Information" link on the MK page (accessible from the RI home page). Refer to this page for additional MK7.3 information that may have been added after these release notes were prepared. The complete URL is:

http://www.opengroup.org/RI/PubProjPgs/MK73.htm

2.2 Reference Platform Definition

The following list constitutes the supported hardware configurations for MK systems release 7.0 and later (as well as AD systems release 3.0

and later). There are significant differences for earlier MK and AD systems. The software described by these release notes is designed for (and has been tested on) these configurations.

There are required minimum configurations, recommended minimum configurations, and obsolete configurations. The obsolete configurations are believed to work but have not necessarily been tested recently. If an obsolete configuration does not work, we may decide not to fix the problem.

2.2.1 System

PC/AT compatible with i486 or Pentium processor and ISA/EISA/PCI bus. Hardware floating point is required (486DX for 486 machines).

Obsolete: PC/AT machines based on the i386.

2.2.2 Video/Monitor

Required: VGA display in 25 x 80 character mode

We use color monitors due to historical problems with monochrome monitors.

2.2.3 Memory

MK 7: 16 MB required, 32+ MB required for use with X11, and otherwise recommended. (We only regularly test on machines with 32 MB or more.)

AD 3: 32 MB required, 36+ MB recommended for server development.

2.2.4 Disk Controllers/Host Adapters

The following hardware configurations are supported (only one controller/adapter at a time, however):

SCSI Disk with Adaptec AHA-1542B/1542C adapter (ISA bus)

SCSI Disk with Adaptec AHA-1542CF adapter (ISA bus) (MK 7.0a+)

SCSI Disk with Adaptec AHA-1542CP adapter (ISA bus) (MK 7.2+)

SCSI Disk with Adaptec AHA-1742 adapter (EISA bus)

SCSI Disk with NCR/Symbios 53C810 or 53C825 PCI adapters (825 in narrow SCSI only). Note that 810 adapter boards require the BIOS to be present on the motherboard. (MK 7.2+) (AD 3.0+)

Obsolete:

ESDI Disk with Western Digital 1007 controller

Note: Some IDE drives may work with WD1007-compatible IDE disk controllers.

We strongly suggest SCSI disks in all configurations. That is what we use (almost exclusively) for our own workstations.

SCSI disks are required for use in real-time configurations.

2.2.5 Disk

Currently only one disk may be used at a time.

Required:

300 MB disks for minimal test machines.

600 MB disks on program development machines.

Recommended:

1 GB disks.

2.2.6 Tapes

SCSI DDS-1 (4mm DAT) tape drives. (MK 7.2+ only)

SCSI QIC-150 is known *not* to work on MK 7.2.

2.2.7 Ethernet

Required: The following 10Mbps Ethernet boards are supported:

SMC WD8003/8013 EP, Elite, Elite 16, Ethercard plus family (e.g., SMC 8013EWC, which supports thinnet, thicknet and UTP or SMC 8013EPC which supports thinnet and thicknet).

DEC Etherworks PCI Ethernet card (DE435-AA/DE450). (MK 7.2+ only) (AD 3.1+)

SMC 8216C (EliteUltra) seems to work in a compatibility mode. (MK 7.2+) (AD 3.1+)

Obsolete:

3Com Etherlink II cards (3C503) may work, but cannot be used for initial network loading.

2.2.8 FDDI

Required only for use with FDDI. FDDI does *not* eliminate the requirement for Ethernet. The following FDDI board is supported:

Digital PCI FDDI Single Attach (DEFPA-AA) (MK 7.0a+) (AD 3.1+)

2.2.9 Myrinet

Required only for use with Myrinet. Myrinet does *not* eliminate the requirement for Ethernet. The following Myrinet board is supported:

> Myricom M2-PCI32 and M2F-PCI32 PCI boards. (MK 7.2+ only)

2.2.10 Floppy

One 3.5” (1.44 MB) drive is required as Drive A:.

One 5.25” (1.2 MB) drive is supported as Drive B:.

2.2.11 Graphics Controllers

The X11 window system is supported only on MK systems.

MK uses the XFree86 distribution for X11. The following graphics controllers are supported:

> #9 GXE (currently used in many RI workstations) (make sure you get revision level 12 or higher)
>
> Diamond Stealth PCI 64 bit graphics card, with at least 2 MB of video RAM, S3 Vision-968 controller.

Obsolete:

> Paradise P1A VGA (not been tested recently)
>
> Sigma Legend
>
> Sigma Legend II
>
> Tseng Labs ET4000-based boards

2.2.12 High resolution Monitors

We don’t know that there should be particular issues with the graphics monitors — as long as the monitor is compatible with the graphics controller that you use. The RI happens to use Mitsubishi Diamond Scan 20, NEC Multisync 6FG, and NEC XP21 color monitors.

2.2.13 “In-kernel” CORDS: (MK 7.2+ only)

Only the following LAN cards are supported for use with In-kernel CORDS:

> SMC WD8003/8013 EP, Elite, Elite 16, Ethercard plus family.
>
> SMC 8216C (EliteUltra).
>
> Digital PCI FDDI Single Attach (DEFPA-AA).
>
> Myricom M2-PCI32 and M2F-PCI32 PCI boards.

2.2.14 Miscellaneous

2 RS232 serial ports (COM1 and COM2)

1 parallel port (unidirectional) (MK 7.0a+ only)

Soundblaster (MK 7.2+ only)

3-button bus mouse, using the Logitech Protocol (MK 7.0+ only)

MK is being used at The Open Group Research Institute on the following models of x86 PC/AT compatible systems:

- Digital Celebris XL 5100 — 100 MHz Pentium with PCI/ISA bus, 32+ MB RAM, Diamond Stealth graphics card, and Symbios 53c810 SCSI chip on motherboard.
- Digital DECpc 450ST — 50 MHz i486 with EISA bus, 36+ MB RAM, #9 GXE graphics card, and Adaptec 1742 SCSI adapter.

Most PC/AT (386+) class machines can be expected to work provided the peripherals are in the list of supported equipment presented above.

2.3 MK 7.3 Distribution Tape

The MK7.3 distribution tape contains two *dd*(1) volumes and four *tar*(1) volumes, as shown in Table 1 below.

Volume	Description	Size
1	MK7.3 binary image (root)	12 MB
2	MK7.3 binary image (usr filesystem)	37 MB
3	MK7.3 boot floppy disk images	5 MB
4	MK7.3 documentation	3 MB
5	MK7.3 sources + build logs	39 MB
6	MK7.3 build tools	45 MB

Table 1: Disk Space Requirements

The first three volumes of the tape contain the *MK7.3 binary release*, while the remaining volumes comprise the source distribution.

The first and second volumes of the tape contain compressed *tar* images of the *root* and *usr* filesystems, which were written onto the distribution tape by *dd*. These two binary volumes contain a basic MK7.3 environment with standard commands and libraries. The third volume contains *tar* images of the floppy disks that are used to bootstrap MK7.3 onto a new system.

The *MK7.3 source distribution* consists of the last three volumes of the tape, each of which is in *tar*-format. The fourth volume contains documentation, including these release notes and manual pages. The fifth volume contains the source files for MK7.3 and reference build log files. The last volume provides the (binary) tools needed to build the release.

Be sure that you have enough disk space before attempting to extract the contents. Table 1 shows the size of each volume. If you are only interested in installing the MK7.3 binary distribution, follow the instructions provided in Section 2.4. If you wish to install the source code, build the system, and install the resultant binaries, follow the instructions provided in Section 2.5.

2.4 Binary Installation

The first and second volumes of the tape contain compressed *tar* images of the *root* and *usr* filesystems, which were written onto the distribution tape by *dd*. These two volumes contain the installed image of the basic MK7.3 binary environment. The third volume contains *tar* images of three floppy disks that are used to boot an MK7.3 system. After extracting the floppy images onto your system, you will need to copy them onto blank double-sided high-density 3.5-inch floppy disks (1.44 MB capacity). On a UNIX system, you can use the *dd* command to make a bit-for-bit copy of the file onto the floppy disk device.

The third volume also includes a shareware DOS utility (rawrite) that can be used on any PC to write a file onto a floppy disk. If you are running MK7.X on a system, you can use the *mkbootfloppies* script to create bootable floppy disks for MK7.3. The script can be easily adapted for use on other UNIX systems.

The binary distribution can be installed using either the network or a local SCSI DDS-1 (4mm DAT) tape drive. Throughout this chapter, the machine on which you are going to install MK7.3 is referred to as the *target* machine. If you intend to use the network for installation on the *target* machine, you will need to use a *remote* machine which has either the remote tape drive from which you will read the MK7.3 distribution tape, or the first two volumes of this tape previously unloaded onto a local disk. You will need an account on this *remote* machine with access to the tape drive, or to the location on the local disk where the contents of the tape reside. Moreover you should enable remote access for this account for the *root* user from the *target* machine through *rsh* by modifying the *.rhosts* file in the HOME directory of this account.

Section 2.4.1 describes the steps that may be necessary to prepare a new system for a binary installation. It is not necessary to reformat the hard drive if you are upgrading an existing MK7.X system. Section 2.4.2 steps through the installation process once the preparations have been completed.

2.4.1 Preparation for Binary Installation

In order to use a SCSI drive as the system disk for MK7.3, you must set it as SCSI unit 0. Similarly if you intend to use a local DDS-1 tape drive to install the binary distribution, set the tape drive to SCSI unit 4.

2.4.1.1 Preparing the Target Machine for Network Access

You will need to choose a host name for your *target* machine along with an IP (Internet Protocol) network address. (Please consult with your network administrator). You will also need to know the *netmask* for your local network. Before installing an ISA bus network adapter in your target machine, you should configure the I/O port address, interrupt level and the RAM base address (if applicable) so the network interface can be recognized by the microkernel. See the Ethernet Board Configuration description in Section 2.7.1 for detailed information on how to properly configure your ISA network adapter.

2.4.1.2 Preparing Remote Machine Access

NOTE: Skip this section if you are installing MK7.3 using a local SCSI tape drive.

The *remote* machine is the one from which you will read the MK7.3 tape; so you must know the name of the host and its IP address. We require that the *target* and *remote* machines be connected to the same subnet. On the *remote* machine, you should add an entry in the */etc/hosts* file containing the IP address and hostname of the *target* machine. For instance, if the hostname of the *target* machine is *hal* with the IP address 123.45.67.89, you should have the following entry in the */etc/hosts* file on the *remote* machine:

```
123.45.67.89    hal
```

Similarly, you will need to have an account on the *remote* machine that you can use for access to the SCSI tape drive (or remote disk storage with the tape contents) located on the *remote* machine. In the HOME directory of this account, you should create a *.rhosts* file granting access to the *root* user from the *target* machine. If the hostname of the *target* machine is *hal*, you should have the following entry in the *.rhosts* file of the HOME directory for the account you will use. This user must have read access to the remote tape drive (or remote disk storage with the tape contents) on the *remote* machine:

hal root

You should note the raw device name of the SCSI tape drive (specifically the non-rewinding tape device) or the pathnames for the compressed *root* and *usr* file systems extracted from the tape. Depending upon the type of the *remote* machine, the tape device may have different names, such as:

/dev/nrmt0h, /dev/nrct0, /dev/rmt/0hn, /dev/nrst4, ...

2.4.1.3 Disk Formatting

You may need to perform a low-level "surface format" of the hard disk on your *target* machine. This procedure partitions the disk into sectors and then checks for bad areas of the disk. This process can require up to an hour to complete, depending upon the size of the disk and speed of the PC. Most disk manufacturers (HP, IBM, Seagate, Quantum, etc.) ship hard drives that are preformatted. Also, some vendors will format the disk for you prior to shipment. The documentation you received with your disk, computer, or DOS will explain how to format your disk. The procedures vary widely and are beyond the scope of this document. Please contact the disk drive manufacturer if you have low-level formatting problems.

Note that almost all SCSI disk drives are already preformatted. You do not need to reformat a disk when upgrading an existing MK7.X system.

2.4.2 Binary Installation Process

There are five steps necessary to install the binary MK7.3 distribution, each of which is described below. First you must create a set of bootstrapping floppies and use them to boot the computer (Section 2.4.2.1). Next you need to prepare the hard disk drive for the binary installation (Section 2.4.2.2). If you are installing the release via the network, you will also need to initialize the network adapter (Section 2.4.2.3). Next you can extract the binary images onto the hard disk (Section 2.4.2.4). Finally, you will need to customize certain system configuration files (Section 2.4.2.5).

Installing MK7.3 is intended to be a simple, cookbook process.

In the following paragraphs, all arguments enclosed within square brackets indicate optional parameters.

2.4.2.1 Booting the Target Machine from Floppy Disks

We will assume for the discussion below that you are performing a "simple" installation. In particular, we assume that you do not want to reserve a portion of the disk for other operating systems (such as DOS or Linux), that you do not have excessive bad blocks in inopportune locations (where the system would put the bad block table), and that you do not need to override the disk geometry parameters that are maintained internal to BIOS.

You will use the boot floppies that you created earlier (by copying the contents of the third volume of the MK7.3 distribution tape) to bring up the system in "single user" mode. (See the discussion in Section 2.4 above concerning the creation of these floppy disks.) Depending on the BIOS of your machine, loading from these floppies may be rather slow.

Insert the first floppy and reboot (or power toggle) the machine. The system should print out a few lines — indicating the amount of memory and the bootstrap version — and then print

```
boot:
```

At this prompt, press the "Enter" (or "Return") key. Then, you will see a number printed out, a pause, a "+", etc. While the bootstrap is reading from the floppy, it prints sizes for this microkernel (text size, data size, bss size, symbols size). Next, you will see several lines of printout as the MK7.3 microkernel describes the hardware configuration that it finds.

Shortly after, the bootstrap will prompt:

```
configuration file: /dev/boot_device/mach_servers/boot-
strap.conf
```

Press the "Enter" key after this and the following prompt will appear:

```
default_pager: /dev/boot_device/mach_servers/default_pager
```

Press the "Enter" key again and the next prompt will appear:

```
startup: /dev/boot_device/mach_servers/startup -s -a fd0b
```

At this point, remove the first floppy and insert the second boot floppy. Once you have switched floppies, you can proceed with the boot process by pressing the “Enter” (or “Return”) key. There will be several lines of output as the server starts. Then you will see the following prompt:

```
root device: [fd0b]
```

Remove the second floppy, replace it with the third floppy disk and press the “Enter” key. Finally, you should see the “#” prompt, indicating that the server is running in “single user” mode with root privileges.

If nothing is displayed after the “`boot:`” prompt then there may be a display/graphics adapter problem (e.g. the board may have switched to an SVGA mode depending upon your monitor). In this case, you may need to switch to a standard VGA graphics adapter board to avoid the problem, unless your display adapter board can be configured to run in standard VGA mode.

2.4.2.2 Disk Setup

This step prepares the hard disk on the *target* machine. This is accomplished by executing the *disk_setup* command on the target machine:

```
# disk_setup [ide]
```

If you are using a standard SCSI disk, you only need to type:

```
# disk_setup
```

By default, *disk_setup* assumes that you have a SCSI disk attached to an Adaptec or NCR host adapter from which you wish to boot.

disk_setup is an interactive script that initializes the disk for use under MK7.3. It can optionally perform three disk initialization steps:

- Low-level formatting, which will destroy any/all data on the disk. Using IDE, EIDE or ESDI disks, this procedure takes roughly 15 minutes per 100MB. On a Pentium system using a SCSI disk drive, formatting the disk requires approximately 15 minutes per gigabyte.
- Partitioning the disk (using the *vtoc* utility). You will be asked to specify parameters for each disk partition interactively. Refer to Section 2.7.4 for an example.
- Creating new filesystems with *newfs*. This is required if you reformatted or re-partitioned the disk.

If an earlier MK7.X system was previously installed, it is not necessary to perform any of the three preceding steps when upgrading to MK7.3. (If you choose to, you will lose any information that is currently stored on the disk.) Nonetheless, you should still run *disk_setup* to extract the tools needed to complete the binary installation. If you are upgrading an MK7.X system and wish to preserve disk data, you can answer “no” to each of the disk initialization options described above.

Finally, after initializing the disk as appropriate, *disk_setup* extracts a set of tools from the third boot floppy that will be used to complete the binary installation. (The boot floppies were described earlier in Section 2.4.)

2.4.2.3 Network Setup

NOTE: **Skip to Section 2.4.2.4 if you are installing MK7.3 using a local SCSI tape drive.**

This step defines the network configuration on the *target* machine by means of the *net_start* command:

```
# net_start targ_ip_addr device_name netmask
```

where the *net_start* parameters are defined as:

> `targ_ip_addr` — the Internet address of the *target* machine.
>
> `device_name` — the Ethernet adapter on the *target* machine. The name is *wd0* for the SMC WD8013EP family of ISA boards (nee Western Digital) or *tu0* for the DEC DE-450 “TULIP” PCI Ethernet adapter.
>
> `netmask` — the network mask for the LAN to which the *target* machine is connected (cf IP address of your *target* machine).

For example, assuming your *target* machine IP address is 123.45.67.89, its Ethernet board is an SMC WD8013EP (or SMC Elite8), and the netmask of your class C network is 255.255.255.0, you would type:

```
# net_start 123.45.67.89 wd0 255.255.255.0
```

Note that the length of the netmask field will vary according to the type of network number assigned to your LAN (class A, B or C IP addresses).

2.4.2.4 Download the Binary Images

After having rebooted your machine to single user mode, you should perform the final step of the installation process by issuing the *mk_install* command. This command is used for all binary installations (both local tape and network installations).

If the installation procedure fails during this step (software downloading), you should correct the problem and rerun the *mk_install* script.

Eventually, even if you need to reboot your *target* machine, *mk_install* will be the only step you will need in order to complete the installation process after rebooting with the floppies (unless you provided incorrect parameters when running *net_start* in the previous step, described in Section 2.4.2.3).

Network installations

If you are using a *remote* machine to supply the distribution tape files to be installed on the *target* machine, you should use the following form of the *mk_install* command:

```
# mk_install rem_ip_addr rem_account rem_device
```

where the *mk_install* parameters are defined as:

`rem_ip_addr` — the Internet address of the *remote* machine.

`rem_account` — the username you will use on the *remote* machine.

`rem_device` — the remote non-rewinding tape drive on the *remote* machine that you will use to read the tape. If the `rem_device` value is "`prompt`", then you'll be prompted to provide full pathnames for the volumes you retrieved from the distribution tape onto the local disk of the remote machine. These two compressed *tar* files contain the *root* file system, and the *usr* file system.

Thus, if the IP address of the *remote* machine is 123.45.67.89, the account you will use on it is *root* (the account you use must permit read access to the tape drive, or to the local storage containing the binary images), and the no-rewind tape drive device is */dev/nrst4*, then you would type:

```
# mk_install 123.45.67.89 root /dev/nrst4
```

If the binary images reside on a local disk of the *remote* machine then use:

```
# mk_install 123.45.67.89 root prompt
```

You will then be prompted for the absolute pathname on the remote system of the compressed *tar* file containing the *root* file system, and later for the file containing the *usr* file system.

If this step fails using a network install, it is most likely for one of the following reasons:

- you cannot access the *remote* machine
- access is not granted in the *.rhosts* file of the account used to access the *remote* machine

- you specified an invalid remote special device name or incorrect pathname to the *install* images.

If a failure occurs and the tape has been accessed (e.g., a failure occurs during the installation process), remember to rewind the tape before resuming the installation process (e.g., by using the *mt(1)* command).

Local tape drive installations

If you use a local SCSI tape drive to read the distribution tape on the *target* machine, you should insert the tape in the tape drive, wait for the tape to rewind, and then type:

```
# mk_install local
```

Note that the *mk_install* script requires the tape drive to be set for SCSI unit 4. You should see the names of the loaded files displayed on the console as the installation proceeds.

If this step fails using a local DDS-1 SCSI tape, you should verify that the SCSI tape drive is set as SCSI unit 4.

Also, if the tape has been accessed and a failure occurs during the installation process, remember to rewind the tape before restarting the installation process (e.g., by using the *mt(1)* command).

2.4.2.5 Reboot the Target Machine

When software downloading is completed, use *sync* to update the contents of the hard disk on the *target* machine. Then issue a *reboot* command to restart the *target* machine. Once the reboot has begun, remove the floppy from the floppy drive so that your machine will boot from the hard disk.

After the reboot you need to login as *root*. The first time you boot your target machine with MK7.3 you must run the */postboot* script. In addition, standard customization is necessary, and you will need to modify the following site-dependent files:

- */etc/hosts* — to set at least the IP address of the target machine
- */etc/named.boot*, */var/named/hesiod.ca* — for the DNS nameserver
- */etc/resolv.conf*
- */etc/rc.config* — to set the target machine hostname, IP address, netmask, network interface and gateway variables
- */etc/passwd*
- */etc/group*
- */etc/printcap* — to specify network printers and print servers

- */sbin/init.d/inet* — to configure your local network IP and gateway
- */etc/ntp.hosts*, */var/adm/ntp/ntp.conf* — for network time protocol configuration
- */var/adm/sendmail/sendmail.cf* — for sendmail configuration

If you are upgrading an existing MK7.X system, most of these local files will already have been configured.

If the hard disk is larger than 300 MB, there will be an additional partition (*/dev/sd0e* for SCSI, or */dev/hd0e* otherwise) that you may use. You must run the *newfs* command on this partition before it can be mounted as an additional file system. If you want to mount it automatically at boot time, just add an entry to the */etc/fstab* file.

2.5 Source Installation

The MK7.3 source distribution consists of the last 3 volumes of the tape. The fourth volume contains documentation including the MK7.3 release notes and manual pages. The fifth volume contains the source files for MK7.3 and reference-build log files. The last volume provides the (binary) tools needed to build the release.

2.5.1 Extracting Files from the Distribution Tape

Be sure that you have enough disk space before attempting to extract the contents. Table 1 shows the size of each tape volume, while Table 2 lists the approximate disk space required to build the MK7.3 microkernel and server. (The exact size will vary depending upon the configuration selected).

Description	Size
MK7.3 microkernel (object tree)	5 - 7MB
OSF/1 server (object tree)	8 - 10 MB

Table 2: Disk Space Needed for Objects

In order to read the MK7.3 distribution, *cd* to the intended build directory and extract the desired volumes using the non-rewinding tape device (e.g., */dev/nrst4* for a DDS-1 (4mm DAT) drive set for SCSI unit 4 using MK7.3):

$ `cd` *<build_path>*

$ `tar xvf` *non-rewinding-tape-device*

Repeat the *tar* command to extract volumes four through six from the tape. (For some tape devices, you may need to issue an extra *tar* command to skip beyond the end of the current volume).

For the remainder of this section, assume that the source tree has been extracted and placed under the *./mk7.3* subdirectory of the desired build directory:

<build_path>/mk7.3

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

The extracted MK7.3 tree will contain the following subdirectories:

./doc – directory containing release-related documentation

./export – tree for header files & libraries needed when building

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

./src – tree containing source code

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

./logs – reference build logs for source (as built by the RI)

./boot - bootstrap floppy disk images

The source code provided under *./src* includes only the source files required to build the MK7.3 microkernel and server, the associated software needed for the system to boot, plus related MK commands and utilities. Note that the source tree does not include those commands and libraries that are unchanged from the OSF/1 1.3 release.

2.5.2 Building the Release

This build process should work on a DEC PC/AT class machine running MK7.2 or later release (see Section 2.2 above). The *./src* tree contains all the source files needed to build the microkernel, server and related MK utilities. However the MK7.3 source distribution does not include source code for the entire set of commands and libraries from OSF/1 1.3. In order to build the full collection of UNIX commands and libraries, you will require the complete OSF/1 source release and should contact The Open Group Direct (direct@opengroup.org).

The remainder of this section assumes that the contents of the source archive reside under the directory *<build_path>/mk7.3* as described above.

There are a few simple steps to follow when building the source tree. The build process is divided into steps that create MK7.3 components in the required order.

In order to build a complete bootable MK7.3 system, you must build all of the following system components (listed with their default pathnames):

Primary bootstrap	*/stand/boot*
Secondary bootstrap	*/mach_boot*
Bootstrap task	*/mach_servers/bootstrap*
MK microkernel	*/mach_kernel*
Default pager	*/mach_servers/default_pager*
Name Server	*/mach_servers/name_server*
MK Server	*/mach_servers/startup*

There are a variety of different kernel and server configurations you can build, and you should be sure you understand the ramifications of the options you select. The steps below describe the build procedure for compiling the entire MK7.3 system.

If any step fails in the build process below, it is important to identify the problem, fix the error, and rebuild, before proceeding to the next step. Any errors will appear in the log files preceded by the string "*** Error". Reference logs from each step (as built by the RI) are provided under *<build_path>/mk7.3/logs*. Although pathnames to the build tree and tools will differ, you can generally compare the reference logs to the output from your build logs below.

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

Utilities:

cc
lex

yacc

Libraries:

libc.a
libl.a
libln.a

The following steps assume you're using a *sh*(1)-like shell; the example target machine to build for is a PC/AT platform running the MK7.3 binary distribution.

Sources for the FSF *gcc* and the ODE minimal tools set are *not* included here. Only the binaries for a PC/AT platform running an MK-based OSF/1 are included (in volume 6 of the distribution tape). If you need to re-build or port these tools for different platforms, please request a tools distribution from The Open Group Research Institute.

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

```
$ PATH="<build_path>/mk7.3/tools/at386/hostbin:$PATH"
$ export PATH
```

2. Copy the build.sh script from the setup area, and build header and MIG definition files:

```
$ cd <build_path>/mk7.3/src
$ cp setup/AT386/build.sh .
$ chmod +x build.sh
```

This step populates the *./export* subtree with header files:

```
$ sh -f build.sh AT386 osf1 . \
  MAKEFILE_PASS=FIRST 2>&1 | tee first.log
```

The reference log is *<build_path>/mk7.3/logs/first.log*.

3. Build the microkernel executable file:

The default configuration for the MK7.3 microkernel is *PRODUCTION*. You can change this configuration by setting the environment variable *MACH_KERNEL_CONFIG* to one of the other available configurations before running *build.sh*. Common configurations include:

- *PRODUCTION* – maximal performance, with no debugging capabilities.
- *FAST* – almost-comparable performance, but with kernel debugger (*ddb*) and all task symbol tables provided.

- *FAST+RT* – development-oriented, with real-time features enabled.
- *FAST+RT+ETAP* – like *FAST+RT*, but with ETAP-related code enabled.

Those configurations that include "RT" in their name are referred to as real-time configurations. They provide kernel features, such as kernel preemption, that are intended to support real-time applications. (N.B.: Currently preemption only works on uniprocessor systems).

```
$ cd <build_path>/mk7.3/src
```

```
$ sh -f build.sh AT386 osf1 \
  mach_kernel build_all 2>&1 | tee ukernel.log
```

The reference logs are *<build_path>/mk7.3/logs/ukernel*.log*.

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

```
$ cd <build_path>/mk7.3/src
```

```
$ sh -f build.sh AT386 osf1 . \
  MAKEFILE_PASS=SECOND 2>&1 | tee second.log
```

After this step, you should find the following files under *<build_path>/mk7.3/export/at386/usr/ccs/lib*:

```
$ ls <build_path>/mk7.3/export/at386/usr/ccs/lib
```

```
crt0.o          libgipc.a       libpthread.a
gcrt0.o         libm.a          librthreads.a
libboot.a       libmach.a       libsa_fs.a
libc.a          libmach_p.a     libsa_mach.a
libcthreads.a   libmachid.a     libservice.a
libcurses.a     libnetmemory.a  mcrt0.o
libflipc.a         libnetname.a
```

(There is a separate subdirectory named *posix_draft_4* that contains an earlier version of the POSIX threads library, *libpthreads.a*).

This step will also build all necessary libraries and exported header files. The reference log is *<build_path>/mk7.3/logs/second.log*.

5. Build the bootstrap task:

```
$ cd <build_path>/mk7.3/src
```

```
$ sh -f build.sh AT386 osf1 \
  bootstrap build_all 2>&1 | tee bootstrap.log
```

The reference log is *<build_path>/mk7.3/logs/bootstrap.log*.

6. Build the default_pager:

```
$ cd <build_path>/mk7.3/src
```

```
$ sh -f build.sh AT386 osf1 \
  default_pager build_all 2>&1 | tee pager.log
```

The reference log is *<build_path>/mk7.3/logs/pager.log*.

7. Build all remaining MK commands and services:

```
$ cd <build_path>/mk7.3/src
```

```
$ sh -f build.sh AT386 osf1 \
mach_services build_all 2>&1 | tee services.log
```

The reference log is *<build_path>/mk7.3/logs/services.log*.

8. Build the OSF/1 server:

The default configurations for the MK7.3 server are *PRODUCTION* and *DEBUG*. You can change the configuration by setting the environment variable *SERVER_CONFIG* to one of the alternate configurations before running *build.sh*:

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

```
$ cd <build_path>/mk7.3/src
```

```
$ sh -f build.sh AT386 osf1 \
  osf1_server build_all 2>&1 | tee server.log
```

Note that by default the server is built in two flavors: both collocated, as *osf1_server.<config>.INK* (loadable into the kernel address space) with an *.INK suffix, and as *osf1_server.<config>* built to run in user space.

Also note that the *FAST* and *PRODUCTION* configurations are currently identical, although this may change in future releases. The reference logs are *<build_path>/mk7.3/logs/server*.log*.

9. Build the primary and secondary boot programs:

```
$ cd <build_path>/mk7.3/src
```

```
$ sh -f build.sh AT386 osf1 \
stand build_all 2>&1 | tee stand.log
```

The reference log is *<build_path>/mk7.3/logs/stand.log*.

Copy the microkernel image to the root directory with a name that identifies the kernel configuration. See the next section for directions on how to boot this image on a machine.

2.5.3 Installing and Booting the Built Images

You must install the microkernel and server images built during the previous steps before booting them. We suggest giving images names that reflect the configuration with which they were built, and the purpose for which the build was done. For instance, you might copy the bootable kernel produced into the root directory as */mach_kernel.MK7.FAST*, and copy the collocated server produced into the */mach_servers* directory as *osf1_server.MK7.FAST.INK*.

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

```
boot: mach_kernel.MK7.FAST -- /mach_servers/bootstrap
```

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

```
$ ln /mach_kernel.MK7.FAST /mach_kernel
$ ln -s /mach_servers/osf1_server.MK7.FAST.INK \
  /mach_servers/startup.INK
```

You might also create the file */mach_servers/bootstrap.conf*. After the boot loader loads the kernel and the bootstrap task, the kernel is given control and launches the pre-loaded bootstrap task as the first task in the system. The bootstrap task, in turn, is responsible for loading and starting additional servers. The set of servers it starts are specified in the *bootstrap configuration file*, whose default pathname is */mach_servers/bootstrap.conf* on the *root* filesystem. The bootstrap configuration file allows you to customize the file names and options for each server that gets started as part of the normal boot process.

The following example shows a typical *bootstrap.conf* file for an MK7.3 configuration, using an in-kernel OSF/1 (UNIX) server:

name_server /mach_servers/name_server
def_pager /mach_servers/default_pager sd0b
-k startup /mach_servers/osf1_server.INK sd0a

Boot Options

During the execution of the bootblock code, the console displays the secondary boot prompt (`boot:`) and pauses for several seconds. By typing a space character (“ ”) at the console at this time, you can cause the bootstrap to suspend execution of its default parameters and to wait for a specific boot command to be entered. This allows the user to override the default parameters, which boot the microkernel named */mach_kernel*, and start the server named */mach_servers/startup*.

The boot command has the following general syntax:

disk(unit, partition)/kernel [k-options] bootstrap [b-options]

where:

a. *disk* is either *hd* for AT, IDE, and ESDI disks or *sd* for SCSI disks
b. *unit* is the logical disk unit number from which you want to boot (usually 0 or 1; default is 0).
c. *partition* is the partition containing the root file system (value between *a* to *p*; default is *a*).
d. *kernel* is the name of the file on the root partition containing the microkernel binary you want to boot (default is *mach_kernel*). If this name contains no pathname prefix, it will be searched for in the root directory.
e. *k-options* is a set of options for the kernel, including:

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

   -*s* — single-user mode (passed to server)

   -- — indicates end of kernel options
f. *bootstrap* is the name of the file on the root partition containing the bootstrap task binary you want to boot (default is */mach_servers/bootstrap*). If this name contains no pathname prefix, it will be searched for in the root directory.
g. *b-options* is a set of options for the bootstrap task, including:

   -*a* — ask for the name of the bootstrap.conf file. This option permits non-default servers to be used as name server, default pager, and OS personality.
h. Every field is optional.

For example, if you want to have the boot program prompt you for the names of all configuration files and servers, at the boot prompt, type:

```
boot: -- -a
```

The bootstrap will prompt for the configuration file and each server, printing a default and allowing you to edit each filename. So, for example, to boot files located in the */mach_srv.old* directory (user input shown in bold):

```
boot:-- -a
Loading sd(0,a)/mach_kernel
583059+121020+172244[+59728+59135]
Loading sd(0,a)/mach_servers/bootstrap
90331+5708+204[+12000+10970]

...

Configuration file:/dev/boot_device/mach_srv.old/bootstrap.conf
name_server: /dev/boot_device/mach_srv.old/name_server
default_pager: /dev/boot_device/mach_srv.old/default_pager
startup: /dev/boot_device/mach_srv.old/osf1_server.FAST.INK
```

2.6 Common Problems

Here are some trouble-shooting hints should problems arise during the installation process:

- Unless you are running X11, MK will use a "monochrome 80 x 25" mode character display. Some VGA cards misbehave; you might get better results with a simple monochrome board if you have one. Also, verify that you have initialized your VGA board for this mode; some boards require special setup.
- If you do not have direct access to a UNIX machine with a tape drive, you should transfer (by whatever means you have) the first two volumes of the MK7.3 distribution tape to some disk location on a UNIX system. (These two volumes are copies — written by *dd* — containing compressed *tar* images of the *root* and *usr* filesystems, respectively.) This machine will then become the *remote* machine, and in the *mk_install* command, you can use the *prompt* option, and give pathnames for the extracted compressed *tar* files.

  For instance, on an MK7.X system with a DDS-1 (4mm DAT) tape drive set to SCSI unit 4, you could copy the binary volumes to disk using the *dd* command:

  ```
  $ dd if=/dev/nrst4 of=root.tar.Z
  $ dd if=/dev/nrst4 of=usr.tar.Z
  ```

2.7 Supplemental Information

The final section of this chapter provides detailed information that you may require for configuration and debugging during installation.

2.7.1 Ethernet Board Configuration

This section specifies the proper configuration of the Western Digital(SMC) WD8003/WD8013 EP/WD8216C family of ISA bus network adapter cards for use in MK systems.

You should select 0x280 as I/O port address and IRQ 2 for the interrupt level; the dual RAM base address should be located at 0xD0000. Note that IRQ 2 and IRQ 9 are equivalent for the PC/AT platform. The microkernel also probes for a network adapter at these alternate settings: port 0x240, RAM address 0xCC000 and IRQ 5 (device wd1), and port 0x300, RAM address 0xD8000 and IRQ 5 (device wd0). The valid settings are listed in Table 3 below:

Device	IRQ	I/O base	RAM address
wd0	2 / 9	0x280	0xD0000
wd0	5	0x300	0xD8000
wd1	5	0x240	0xCC000

Table 3: ISA Ethernet adapter settings

It may be necessary to first change these settings for the network adapter using the (DOS) setup utility program that came with the board. The factory default values for SMC / Western Digital network adapters are *not* the proper configuration for use with MK7.3. You should make sure the board jumpers are set for “software” configuration, and then run EZStart or EZSetup to change the default settings.

2.7.2 Debugging {with} MK

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

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

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

db> *x/<flag>t unix::<varname> $task<n>.<p>*

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

where:

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

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

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

2.7.3 MK Paging

MK uses a paging partition that can be specified in either of two files:

- As a command line argument to the microkernel default pager, as defined in */mach_servers/bootstrap.conf* (e.g., *"sd0b"*)
- Or as a "sw" entry in the */etc/fstab* (used by the *swapon* command)

Any use of anonymous *mmap* will be backed by the vnode pager of the OSF/1 server. Temporary memory created by MK microkernel services, such as *vm_allocate* and *copy-on-write*, will be backed by the default pager.

Since shared-library programs are loaded by use of *mmap*, they will have their memory backed by the vnode pager. Statically bound programs will be backed by the default pager.

2.7.4 Sample Disk Partition Table

The *vtoc* utility is used to partition and format the disk. The MK7.3 binary distribution installation scripts assume that the disk uses standard partition assignments. See Table 4 below for a description of the disk partitions.

Partition	Contents	Overlap	Minimum size
a	*root* filesystem	c	40Mb
b	paging	c	80Mb
c	entire disk	all	350Mb
d	*usr* filesystem	c	100Mb
e	optional 2nd *root*	c	0
f	optional 2nd *usr*	c	0

Table 4: Disk Partitions

By convention, the *b* partition is used for paging space. The minimum size of this partition depends upon the amount of physical memory configured on the system. For reasonable virtual memory performance, we recommend that the size of the paging partition should be at least three to four times the size of memory.

The *vtoc* command for a typical MK7.3 workstation configuration might report the following partition sizes:

```
The static data for the vtoc is:
     cyls 3980/ heads 5/ sectors 104; alt ptr 15360 alt len 2048
     4 UNIX partition entries.
a: start 1040, end 1332239, size 1331200 (650 Meg), flag 200, tag 2
b: start 1332240, end 1639439, size 307200 (150 Meg), flag 201, tag 4
c: start 1, end 2069599, size 2069599 (1010 Meg), flag 200, tag 5
d: start 1639440, end 2049039, size 409600 (200 Meg), flag 201, tag 4
```

3 MK7.3 Known Problems

3.1 Introduction

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

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

3.2 Kernel

- DIPC loopback tests do not pass when KKT/Eth is being used. The kernel panics when an rpc local to a node is attempted. [2413]
- The kernel task swapper uses poor criteria for choosing tasks to swap out. It can swap out the only running processes in the system as configured. This is because it treats as "active", any task that is either running or waiting. With this algorithm, it can swap out all the truly active tasks, leaving an idle shell swapped in. [2381]
- Panic: Cthread wakeup occurs under a heavy system load. An activation is allowed to run before the cthread wired port is set. [2379]

- The swapout thread attempts to do work as soon as the thread is started (boot time). As a result of this it garbage collects the system even though at boot time the load is light. [2376]
- The getrusage system call page fault statistics are always set to 0. [2362]
- When trying to open a file with a pathname longer than the maximum pathname length, the error EFAULT is returned. It should be returning the error ENAMETOOLONG. [2356]
- Assertion failure in *task_info()* “Assertion failed: file “task.c”, line 1629: thread”. Improperly failing in cases where the thread does not yet have a shuttle. [2350]
- System hangs due to a race between thread creation and task swapping. If a task is marked for swapout while it is creating new threads, the new threads will not be marked for swapout. [2337]
- The ZONE_DEBUG option which is enabled under DEBUG mach kernels doesn’t adjust the maximum size of a zone with the added fields inserted to maintain a queue of allocated zone entries. This makes the zone artificially smaller since it can contain fewer elements for the given maximum size. A simple test that attempts to allocate TASK_MAX tasks under a DEBUG kernel will fail due to this. [2330]
- Panic: ipc_task_init when running mach_perf task_perf test. This is due to the reaper_thread not running often enough to maintain a pool of free tasks. [2329]
- SCSI subsystem can’t be unconfigured in config.devices. The kernel fails to link due to many undefined references. [2314]
- memory_object_set_attributes_common causes a memory leak if MACH_PAGEMAP and NORMA_VM are defined. [2312]
- Vm_fault will sometimes sleep interruptibly. This should not occur since vm_fault can be called with locks held. If the system sleeps interruptibly it can deadlock with threads waiting on the lock held by a suspended or swapped out thread. [2290]
- Unresolvable page fault in kernel space. An ordinary page fault (probably just to restore a displaced mapping) on a server buffer is not being serviced properly. The problem is that the kernel addresses which map this memory have been removed, not just from the pte, but from the kernel map itself. The problem is caused by code in the I/O error return path. In this particular dump, it appears that there were a couple I/O errors which preceded the crash. Inspection of the ds_read_done routine indicates that the an incorrect memory free may occur in the error path for I/O which is targeted to the colocated server address space.
- At least 2 MIG .defs files (mach.defs and mach_host.defs) have too many interfaces. This causes some overlaps in the msgh_id of different

messages. For example, *task_set_port_space()* overlaps with *processor_set_destroy()* and *device_map()*. [2274]

- Panic "aha U/OVRN on scsi command" occurs on system where rboot was written to the disk with dd command without the conv=sync option. [2269]
- Heavy I/O over a serial lines may result in the line hanging. Output may be resumed with a CTRL-C. This occurs when software flow control is enabled and *ttyinput()* discards a CTRL-Q character because the input queue is full. [2226]
- System panics when page frames that should have been initialized in *pmap_startup()* are found to be uninitialized memory when referenced by *vm_page_release()*. [2225]
- When running CORDS udpiptest with all user space protocols over Myrinet with a FAST+RT kernel, about 2 milliseconds of jitter is seen in the elapsed times. No jitter is seen with a FAST kernel, or with protocols in the kernel space. [2222]
- A CORDS +net+test kernel will only work properly if the system is in single user when x_client or x_server is set to 1 to bootstrap the cords protocol stack. This is due to a limitation of the BOOTP protocol, which only works in kernel mode iff it is the first protocol to use the network device. [2221]
- *Scsi_probe()* calls from the NCR 53C810 driver do not detect a DEC RZ26L disk drive from an MK7.3 DEBUG kernel. The drive is detected from a PRODUCTION kernel. An MK7.3 DEBUG kernel can detect the drive if breakpoints are set in the right places, so I suspect the problem is either timing related or an uninitialized pointer. [2132]
- When running a collocated server, virtual addresses given by a Unix process are not checked in copyin & copyout routines. Unix processes can corrupt kernel address space, resulting in panics and hangs. [2119]
- While working with Mach aware gdb the kernel died with a bus error. The apparent cause of the bus error was that *task_by_pid()* called mach_port_deallocate with a "name" parameter of -1. [2114]
- VSX cfset{i,o}speed tests fail at 19200 and 38400 baud. This may be a symptom of the fact that the driver (mach_kernel/i386/AT386/com.c) doesn't make use of any buffering supplied by the chip. [2017]
- Test 2 of the RPC chains and alert tests (rpctest_test_suite.sh) sometimes results in an assertion failure: file "/afs/ri/project/kernel/build/mainline/src/mach_kernel/ipc/ipc_right.c", line 651: port->ip_srights > 0. Test 1 of the same suite hangs on a 133 MHz system, but not at 100 MHz. The rpc_client/rpc_server cases tests sometimes fail with the message "rpc_client: port_array_out: badptr" when started at case "a". [1976]

- There is a vm_priv inversion in the zone package. The package makes sure that no more then one thread expands a zone at a time. If a non vm_priv thread hits a zone needed in the pageout path and tries to expand the zone, and the system is out of memory, we will deadlock. The non vm_priv thread will take the “expanding” token and the vm_priv thread will come in later and block waiting for the non vm_priv threads to complete. It never will. [1886]
- When running mach_perf in multiuser mode with other processes simultaneously (such as build), mach_perf will do a task_terminate which will send dead name notifications to the server from ports belonging to other Unix processes, which are therefore killed by the server with a signal 9. [1870]
- When removing the -k argument from the bootstrap.conf, the OSF/1 server crashes in old_get_config_info because it tries to access bad memory from argv. This is due to the bootstrap_argument MIG interface which gets a zero in the argument pointer field. The size argument is ok, so it might be a out of line memory passing problem. [1869]
- This appears to be another tty/mouse bugs. When typing in a vi xterm window (started from the Reply button of exmh), echoing suddenly became very slow (about 1 second). The mouse was frozen too, although the second hand of xclock was still ticking. The machine could be pinged, but neither rlogin nor telnet was successful (telnet said it connected). CTL-ALT-BKSP was used to kill the X11R6 server. It restored VGA text mode successfully, but the shell on the console was not responding. Broke into the debugger with CTL-ALT-D and found that the system was completely idle. Most threads in the server were stuck in either getblk (due to a failed try_lock) or an AFS function called Lock_Acquire. [1846]
- VSE fails when doing the truncate and ftruncate tests. The tests involves setting a file-size limit and then attempting to truncate a file upward in size beyond the file size limit. The test fails when the server posts a signal 25 for exceeding file size. POSIX says it shouldn’t do this on a truncate. [1843]
- The error message ‘vnode_pager_change_attributes: timed out after ten seconds’ occurs one to several times a day. When it does, a ten second plus delay in interactivity occurs. The error message does not indicate a fatal problem; it just shows that there would have been a deadlock while AFS was trying to reclaim inodes. If it fails, it will try again. A “large” AFS cache configuration will reduce the frequency of the messages. [1832]
- When running with *vm_remap()* disabled, the server uses *vm_write()* and *vm_read_overwrite()* to perform *copyin()*/*copyout()*. Unfortunately, the implementation of these interfaces uses virtual copies, not physi-

cal copies, yet the COW faults are taken 100% of the time, resulting in both a performance pessimization and a virtual space leak. [1018]

- The pmap_create function should single-thread the memory chunk allocator to avoid multiple allocations. This bug is merely wasteful and then only if NCPUS > 1, hence it is at a low priority. [126]
- The fix made for mk6 CR59 to make the short-circuited RPC path honor a MIG_NO_REPLY return from the server will need to be extended once *thread_abort()* works on migrated threads. [120]
- The Adaptec 154x driver is not intelligent about how it copies buffers from the low 16MB of memory to higher memory on EISA-bus machines that can support more than 16MB. Note that this problem will not manifest itself when using Adaptec 174x board's operating in enhanced mode. [96]

3.3 MK Server

- OSF1 server cannot be built without STREAMS. Removing the STREAMS option plus all its suboptions yields numerous undefined symbols. [2395]
- The default_pager, though multi-threaded and configured to use asynchronous writes to the paging partition, winds up running single-threaded and with synchronous write operations. This is because it gives up its mapping to the page before starting an asynchronous write. The unmapping will not complete until the I/O is complete, so the default pager just hangs in the object termination code waiting for paging in progress to go to 0. The effect is to turn an async write into a synchronous write. While it is waiting, the kernel holds the map lock for the address space of the default pager. This has the effect of blocking other default pager threads that wish to allocate or deallocate memory. [2387]
- If a second drive is specified in /etc/fstab for automounting, the system will fail to properly mount that or the primary drive most of the time. An fsck failure is reported on one of the partitions. If it happens to be the root partition, the system is essentially dead. A workaround is to delete the second disk from /etc/fstab and mount it manually after the system is in multiuser mode. [2276]
- With the Celebris GL 5100 integrated audio devices enabled, problems occur using X11R6 and selfhost sup collection three. The XFree86 server appears to hang (blank screen with frozen cursor) when creating certain new windows. The hanging occurred with the audio configured for IRQ 5 and either DMA channel 0 or 1. [2272]
- The osf1_server.FAST+AFS.INK server appears to use the wrong default root device. If /mach_servers/bootstrap.conf does not specify a startup command line option, the single server claims to be using “ma-

jor=5, minor=0" as the root device. If the startup line contains an explicit root device (e.g. "-k startup startup.INK sd0a") then the server uses the correct root device (major=3, minor=0). [2271]

- Pthread mutex deadlocks, uxthread_yield should release the uniproc lock and call thread_switch with SWITCH_OPTION_DEPRESS. [2189]
- Breakpoints set with gdb in a user application at *thread_switch()* stop at the breakpoint but hang when trying to continue from the breakpoint. This is the ptrace version of gdb, rev 4.16. [2186]
- Data files in an AFS file system are sometimes overwritten with blocks of nulls. [2133]
- In an AFS directory, overwriting an existing executable file often results in a file which fails to execute and instead produces a failure such as: "rtld main: Load of "mut" failed: Exec format error". File "mut" had been overwritten after a small change in the source file. If mut is deleted and the same link done again, it executes ok. [1981]
- System hangs with the message "can't map region (error = 3)". What seems to have happened is that a vm_map done in elf_getxfile failed, so the process was killed. When the process called exit, it eventually got to exec_ungetxfile, which noticed that the SPAGV flag was set and tried to release a reference on the text vnode. However, the text vnode was never set up. [1978]
- VSX test c_iflag failures when ISTRIP or BRKINT flag is set. [1955]
- On a system running X, server panics produce a blank screen with no input possible. [1934]
- Call to *sleep()* from a cthreads child never returns. [1914]
- It is possible to mount an NFS filesystem to a mountpoint and then mount a local device on top of it. For example, I mounted a remote machine's /usr partition as /mnt, and then mounted /dev/fd0 on top of it. It is not possible to mount an NFS partition on top of an existing NFS mount point. [1893]
- The uniproc mutex is not release and then taken back when invoking device_read/write in *disk_read/write()*. Thus server activity is blocked when waiting for I/O to complete. [1809]
- When "ps ax" is used, the server calls thread_info to get information about each thread of each process. Sometimes (perhaps if processes are exiting at the same time ps is invoked), it passes a bad port name as the target thread port in thread_info. In the case analyzed, this port was one for which the (kernel-loaded) server was the receiver. This can crash the kernel or server, because the kernel will believe that the port name is valid, as long as it is not null. We have a collocation design defect here, in that dead-name usage by kernel-loaded servers cannot be

detected. We also have a server bug, because our server supposedly never tries to use a dead name, as long as no task or thread is externally terminated without its knowledge. The server bug may be a race between ps and process teardown. [653]

- There is a non-deterministic assertion failure in the server. Occasionally the server runs into an assertion failure when booting; apparently a port is being reused. [483]
- /dev/zero doesn't exist. The version in AD2 should be adopted. [277]
- Server priority depression incompatible with the new version of C-threads. In the new version, blocked C-threads no longer really block, they just depress their priority (with no timeout). Any server thread that depresses itself temporarily allow a different, unblocked server thread to run, will be put in the same priority run queue as all the blocked threads. Even if it is able to make forward progress almost immediately, it will not get any CPU time until all the blocked C-threads in front of it get their time first. [135]

3.4 Commands

- The df command uses signed ints for block counts. [2460]
- Ping 255.255.255.255 doesn't work. [2361]
- The netstat command -I option returns "Interface not found" even though it can find the interface with different options (e.g. -i). [2326]
- The latest version of tcsh (6.06) sets the OSTYPE environment variable. However, the setting of OSTYPE interferes with the normal operation of the ODE tools. [2241]
- The -prune option to "find" doesn't appear to work properly. If a directory is given as the argument to the -prune switch, find will descend into that directory. [2183]
- Running "ical" fails with: X Error of failed request: BadAlloc (insufficient resources for operation). Note that the ical window shows up and immediately vanishes. Failure occurs with X server in 16 bits per pixel mode. [2147]
- After a workstation has been up quite a while (a week or more), suddenly the X server refuses to create any new windows, claiming that the client is not authorized to connect to the server. Logging out and back in (i.e. restarting X) cures the problem. [2146]
- The newaliases command prints the error message "cannot alias non-local names", while trying to alias local names. For example, the line "postmaster:root" in the aliases file generates this error. [2134]

- The X server crashes with an “Out of memory” error message. Problem can be worked around by adding the switch “-ld 32768” when starting the X server, This will increase X server’s data limit to 32M. [2082]
- If mwm is started without an .mwmrc, the system will dump core. The straightforward workaround is to insure you have a .mwmrc file, even if it is empty. [2073]
- Nslookup works fine normally, but when you ‘set type=any’ or fire it up as ‘nslookup -type=any’ then it fails with a record size incorrect error. [1904]
- Who command prints “tty??” instead of correct tty number. [1849]
- The gdb adapted to understand the new cthreads library should work better with the collocated servers. But in practice any significant use of it causes it to crash out with a memory allocation failure. [614]
- Using the -d option to pinfo (on an ordinary user-mode process) does not work. This is the option that uses *mach_port_space_info()* to query IPC space usage data. [103]

3.5 Libraries

- The cthreads implementation of red zones for cthread stacks wires the red zone memory permanently. This only occurs for cthreads stacks within colocated servers. It was observed that 14 pages get wired in the red zone for each in kernel stack. [2366]
- In the cthreads library the red_zone flag was forced to TRUE, even when the cthread_red_zone_size was not allocated, this leads to a vm_protect failure in stack_alloc. [2188]
- The rthreads library rthread_spawn primitive does not check for error return, and passes a NULL thread id to kernel (to get a ‘wired kernel thread’). [1977]

3.6 Unidentified

- On a two-node cluster, with a STD+WS+NORMA kernel from either NMK17.3 or cnmk_shared, you can reliably cause a multi-user boot of the AD1 server to fail by starting the boot process on both clusters almost simultaneously (node 0 a bit before node 1). So long as you get node 1 to complete registration for process creation before network configuration starts, the boot on node 0 will hang after outputting the hostname of node 0. If you bring up node 0 to multi-user and then add node 1 to the cluster, the failure does not occur. [564]

- When running two servers kernel-loaded, using the -2 option when starting the second server doesn't work. The second server crashes during the printing of the “Root” message. [105]

4 Kernel API Changes

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

a. General evolution of the microkernel interface
b. Collocation for servers, providing a performance enhancement
c. Asynchronous device driver model
d. Scalability improvements
e. Real-Time extensions
f. Event Trace Analysis Package (ETAP)
g. Clocks and timers
h. CORDS and communication support
i. Microkernel support
j. IP Multicast

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

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

Also, as the number of manual pages has increased and the coverage of this chapter has subsumed more than just the kernel interface, newer pages have been marked with a “chapter” number (much as would be found with on-line man pages). In general, these chapter numbers should be interpreted to mean:

1, 8 — user commands issued through UNIX

2 — kernel service calls

3 — library calls

4 — file format and data structure descriptions

The following sections group manual pages by general function. Within each section, pages are presented alphabetically within each chapter, and chapters are presented in turn, starting with chapter one.

Finally, the POSIX threads library is documented on the Web at:

http://www.opengroup.org/RI/OS/MK7/doc/MK73pthreads/posix_threads.html

General Microkernel Interfaces

mach_port_get_attributes

Function — Returns information about a port

SYNOPSIS

kern_return_t **mach_port_get_attributes**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

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

port_info
[out structure] Information about the specified port.

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
name did not denote a right.

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

RELATED INFORMATION

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

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

mach_port_set_attributes

Function — Sets attributes of a port

SYNOPSIS

```
kern_return_t mach_port_set_attributes
        (task_port_t                    task,
         mach_port_name_t               name,
         mach_port_flavor_t             flavor,
         mach_port_info_t               port_info,
         mach_msg_type_number_t         port_info_count);
```

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

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

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

NOTES

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

RETURN VALUE

KERN_INVALID_NAME
: *name* did not denote a right.

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

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

RELATED INFORMATION

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

Data Structures: **mach_port_limits**.

mach_port_status

Structure — Contains information for a port

SYNOPSIS

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

DESCRIPTION

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

FIELDS

mps_pset
: Containing port set.

mps_seqno
: Current sequence number for the port.

mps_mscount
: Make-send count.

mps_msgcount
: Number in the queue.

mps_sorights
: How many send-once rights.

mps_srights
: True if send rights exist.

mps_nsrequest
: True if no-senders requested.

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

NOTES

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

RELATED INFORMATION

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

task_set_port_space

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

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

RELATED INFORMATION

Functions: **mach_port_allocate**.

vm_allocate_cpm(2)

NAME

vm_allocate_cpm - Allocates a region of virtual memory backed by physically contiguous non-pageable memory

SYNOPSIS

```
#include <mach.h>

kern_return_t vm_allocate_cpm(
        host_priv_t        host_priv,
        vm_task_t          target_task,
        vm_address_t       *address,
        vm_size_t          size,
        boolean_t          anywhere);
```

PARAMETERS

host_priv
: The host privilege port policy is kernel patchable. The default microkernel policy is to allow any port value to gain access to this system call. A secure system call could be implemented when the MK kernel **vm_allocate_cpm** system call requires that this port be the privileged host port.

target_task
: The task in whose address space the region is to be allocated.

address
: The starting address for the region. If the address is not at the beginning of a virtual page, the kernel rounds it down to the nearest one. If there is not enough room following the address, the kernel does not allocate the region. The kernel returns the starting address actually used for the allocated region.

size
: The number of bytes to allocate. The kernel rounds this number up to an integral number of virtual pages.

anywhere
: The placement indicator. If FALSE, the kernel allocates the region starting at *address*. If TRUE, the kernel allocates the region wherever enough space is available within the address space. The kernel returns the starting address actually used in *address*.

DESCRIPTION

The **vm_allocate_cpm**() function allocates a region of virtual memory in the specified task's address space backed by physically contiguous non-pageable memory.

By default, the kernel rounds all addresses down to the nearest page boundary and all sizes up to the nearest integral number of pages. The **vm_page_size** global variable contains the virtual page size.

Use the **task_self** macro to return the caller's value for *target_task*. This macro returns the task kernel port for the caller.

Initially, there are no access restrictions on any of the pages of the newly allocated region. Child tasks inherit the new region as a copy. The inherited memory isn't physically contiguous. To establish different protections for the new region, use the **vm_protect**() and **vm_inherit**() calls.

NOTES

For languages other than C, use the **vm_statistics**() and **task_self**() calls to return the task's kernel port (for *target_task*).

A region is a continuous range of addresses bounded by a start address and an end address. Regions consist of pages that have different protection or inheritance characteristics.

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

Note that, if **address* does not start at the beginning of a virtual page, then it will be rounded down to the starting address of the nearest virtual page. Thus, care must be taken if the caller's intent is to allocate a region of memory that contains the addresses **address* and (**address*+*size*-1), when **address* will be subject to rounding.

RETURN VALUES

Upon successful completion, the call returns KERN_SUCCESS, indicating that the new region has been allocated. Otherwise, the new region is not allocated and an error code is returned.

ERRORS

KERN_INVALID_ADDRESS
The specified address is illegal.

KERN_INVALID_ARGUMENT
The task in invalid.

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

Any message send or receive error can also be returned.

RELATED INFORMATION

task_get_special_port(2), **vm_deallocate**(2), **vm_inherit**(2), **vm_protect**(2), **vm_region**(2), **vm_statistics**(2)

Server Collocation

mach_subsystem_create

Function — Register information about an RPC subsystem

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

A port naming the registered subsystem is returned.

PARAMETERS

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

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

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

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

RETURN VALUE

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

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

KERN_INVALID_TASK
The target task is dead.

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

RELATED INFORMATION

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

Structures: **user_subsystem**.

mach_port_allocate_subsystem

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

SYNOPSIS

```
kern_return_t mach_port_allocate_subsystem
        (task_port_t        task,
         mach_port_t        subsys,
         mach_port_t *      name);
```

DESCRIPTION

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

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

PARAMETERS

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

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

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

NOTES

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

RETURN VALUE

KERN_SUCCESS
: The right is allocated.

KERN_INVALID_TASK
: The space is null.

KERN_INVALID_TASK
The space is dead.

KERN_RESOURCE_SHORTAGE
Couldn't allocate memory.

KERN_NO_SPACE
No room in space for another right.

RELATED INFORMATION

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

thread_activation_create

Function — Create a thread activation

SYNOPSIS

kern_return_t **thread_activation_create**

(task_port_t	*task,*
mach_port_t	*RPC_port,*
act_params_t	*act_params,*
thread_act_t*	*new_act*);

DESCRIPTION

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

The new thread activation will begin each RPC using the stack pointer specified within *act_params*.

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

PARAMETERS

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

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

act_params
: [in scalar] Points to a structure of this description:

```
struct thread_act_params {
       vm_offset_t           stack;
       vm_size_t             stack_size;
       entry_function_t      entry_func;
       int                   flavor;
       thread_state_t        act_state;
       unsigned int          act_state_count;
};
```

stack specifies the virtual address in the task to be used as the starting address of the user-level stack.

stack_size specifies the size, in bytes, of the user stack.

entry_func is used as the upcall PC when there is no registered subsystem.

act_state specifies optional thread state parameters for architectures that require them.

act_state_count specifies the size of *act_state*.

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

NOTES

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

mig_stub_routine

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

SYNOPSIS

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


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

DESCRIPTION

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

FIELDS

InHeadP
: Pointer to the header for the request message.

OutHeadP
: Pointer to the header for the reply message.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_descriptor**.

routine_arg_descriptor

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

SYNOPSIS

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

DESCRIPTION

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

FIELDS

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

size
: The element size, in bytes.

count
: The number of elements.

offset
: Offset in the list of routine arguments.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_descriptor, rpc_subsystem**.

routine_descriptor

Structure — Defines the nature of each subsystem routine

SYNOPSIS

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

DESCRIPTION

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

FIELDS

impl_routine
: Server function pointer (code entry point).

stub_routine
: Unmarshalling function pointer.

argc
: Number of argument words.

descr_count
: Number of complex argument descriptors.

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

max_reply_msg
: The maximum size for the reply message.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_arg_descriptor, rpc_subsystem**.

rpc_subsystem

Structure — Defines the set of server routines in a subsystem

SYNOPSIS

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


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

DESCRIPTION

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

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

FIELDS

subsystem
: Reserved for system use.

start
: Minimum routine number.

end
: Maximum routine number (+1).

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

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

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

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

RELATED INFORMATION

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

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

Device Driver

device_get_status

Function — Return the current device status

LIBRARY

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

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

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

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

RETURN VALUE

D_DEVICE_DOWN
Device has been shut down

D_INVALID_OPERATION
The device driver does not export any semaphores.

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

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

RELATED INFORMATION

Functions: **device_set_status**.

Structures: *flavor* dependent.

device_read_async

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_read_async returns only invalid parameter errors.

RELATED INFORMATION

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

device_read_async_inband

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_read_async_inband returns only invalid parameter errors.

RELATED INFORMATION

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

device_read_overwrite_async

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

SYNOPSIS

kern_return_t **device_read_overwrite_async**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

bytes_wanted
: [in scalar] Size of data transfer.

buffer

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

RETURN VALUE

device_read_overwrite_async returns only invalid parameter errors.

RELATED INFORMATION

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

device_write_async

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

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

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

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_write_async returns only invalid parameter errors.

RELATED INFORMATION

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

device_write_async_inband

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

SYNOPSIS

kern_return_t **device_write_async_inband**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

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

bytes_wanted
[in scalar] Size of data transfer.

RETURN VALUE

device_write_async_inband returns only invalid parameter errors.

RELATED INFORMATION

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

io_done_queue_create

Function — Create an io_done_queue kernel object

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

KERN_INVALID_ARGUMENT
The *host* argument is not valid.

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

KERN_SUCCESS
The io_done_queue was created.

RELATED INFORMATION

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

io_done_queue_terminate

Function — Terminate an io_done_queue kernel object

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

io_done_queue_wait

System Trap— Wait on an io_done_queue kernel object

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

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

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

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

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

RELATED INFORMATION

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

Data Structures: **io_done_result**.

io_done_result

Structure — Defines asynchronous I/O completion status

SYNOPSIS

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

DESCRIPTION

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

FIELDS

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

IO_DONE_READ The operation was a device_read_async or device_read_async_inband.

IO_DONE_OVERWRITE The operation was a device_read_overwrite_async.

IO_DONE_WRITE The operation was a device_write_async or device_write_async_inband.

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

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

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

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

qd_count
: The number of bytes read or written.

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

RELATED INFORMATION

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

MultiComputer Support

norma_node_self

Function — Return the node index of the current host

LIBRARY

Not declared anywhere.

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

norma_task_teleport

Function — "Clone" a task on a specified node

LIBRARY

Not declared anywhere.

SYNOPSIS

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

DESCRIPTION

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

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

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

PARAMETERS

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

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

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

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

NOTES

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

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

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

task_set_child_node

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

LIBRARY

Not declared anywhere.

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

Real-Time Support

lock_acquire

Function — Acquire access rights to a lock

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

KERN_SUCCESS
: The lock was acquired.

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

KERN_LOCK_UNSTABLE
: The acquired lock has an unstable state.

KERN_LOCK_SET_DESTROYED
: The specified lock has been destroyed.

RELATED INFORMATION

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

lock_handoff

Function — Hand-off ownership of a lock

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

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

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

KERN_SUCCESS
The lock hand-off was successful.

KERN_LOCK_SET_DESTROYED
The specified lock has been destroyed.

RELATED INFORMATION

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

lock_handoff_accept

Function — Accept a lock hand-off

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

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

RETURN VALUE

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

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

KERN_SUCCESS
The lock hand-off was successful.

KERN_LOCK_UNSTABLE
The acquired lock has an unstable state.

KERN_LOCK_SET_DESTROYED
The specified lock has been destroyed.

RELATED INFORMATION

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

lock_make_stable

Function — Clear a lock's unstable state

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

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

KERN_SUCCESS
The lock's state was stabilized.

KERN_LOCK_SET_DESTROYED
The specified lock has been destroyed.

RELATED INFORMATION

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

lock_release

Function — Release ownership of a lock

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

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

KERN_SUCCESS
: The ownership of the lock was released.

KERN_FAILURE
: The calling thread does not own the lock.

KERN_LOCK_SET_DESTROYED
: The specified lock has been destroyed.

RELATED INFORMATION

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

lock_try

Function — Attempt to acquire access rights to a lock

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

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

KERN_SUCCESS
The lock was acquired.

KERN_LOCK_UNSTABLE
The acquired lock has an unstable state.

KERN_LOCK_SET_DESTROYED
The specified lock has been destroyed.

KERN_LOCK_OWNED
Another thread currently owns the requested lock.

KERN_LOCK_OWNED_SELF
The calling thread already owns the lock.

RELATED INFORMATION

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

lock_set_create

Function — Create a new lock set

LIBRARY

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

SYNOPSIS

kern_return_t **lock_set_create**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

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

SYNC_POLICY_FIXED_PRIORITY
a fixed priority policy for scheduling thread wakeup.

RETURN VALUE

KERN_SUCCESS
: The lock set was created.

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

KERN_RESOURCE_SHORTAGE
The kernel could not allocate the lock set.

RELATED INFORMATION

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

lock_set_destroy

Function — Destroy a lock set and its associated locks

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

task
The task associated with the lock set.

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

RETURN VALUE

KERN_INVALID_ARGUMENT
The specified *lock_set* is invalid.

KERN_SUCCESS
The lock set was destroyed.

RELATED INFORMATION

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

semaphore_create

Function — Create a new semaphore

LIBRARY

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

SYNOPSIS

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

DESCRIPTION

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

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

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

PARAMETERS

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

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

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

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

SYNC_POLICY_FIXED_PRIORITY
a fixed priority policy for scheduling thread wakeup.

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

RETURN VALUE

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

KERN_RESOURCE_SHORTAGE
The kernel could not allocate the semaphore.

KERN_SUCCESS
The semaphore was successfully created.

RELATED INFORMATION

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

semaphore_destroy

Function — Destroy a semaphore

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

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

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

KERN_SEMAPHORE_DESTROYED
: The specified *semaphore* was previously destroyed.

KERN_SUCCESS
: The semaphore was destroyed.

RELATED INFORMATION

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

semaphore_signal

Function — Increments the semaphore count

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

RETURN VALUE

KERN_INVALID_ARGUMENT
The specified semaphore is invalid.

KERN_SEMAPHORE_DESTROYED
The specified semaphore has been destroyed.

KERN_SUCCESS
The semaphore has been signalled.

RELATED INFORMATION

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

semaphore_signal_all

Function — Wakeup all threads blocked on a semaphore

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

KERN_INVALID_ARGUMENT
The specified semaphore is invalid.

KERN_SEMAPHORE_DESTROYED
The specified semaphore has been destroyed.

KERN_SUCCESS
The semaphore has been signalled.

RELATED INFORMATION

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

semaphore_wait

Function — Decrement the semaphore count

SYNOPSIS

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

DESCRIPTION

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

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

PARAMETERS

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

RETURN VALUE

KERN_INVALID_ARGUMENT
: The specified semaphore is invalid.

KERN_SEMAPHORE_DESTROYED
: The specified semaphore has been destroyed.

KERN_SUCCESS
: The semaphore wait operation was successful.

RELATED INFORMATION

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

mach_rpc_trap

System Trap— Real-Time RPC trap

LIBRARY

#include <mach/rpc.h>

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

RETURN VALUE

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

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

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

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

NOTES

This function is not designed for use directly by the user. It is automatically generated by MIG to handle a function call. For a complete description of this functionality, refer to: Burke, Edward, Michael Condict, David Mitchell, Franklin Reynolds, Peter Watkins, William Willcox, "RPC Design for Real-Time Mach", OSF Research Institute, Cambridge, MA, April 1994.

RELATED INFORMATION

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

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

mach_rpc_return_trap

System Trap— Real-Time RPC trap return location

LIBRARY

#include <mach/rpc.h>

SYNOPSIS

kern_return_t **mach_rpc_return_trap**
( );

DESCRIPTION

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

PARAMETERS

NONE.

RETURN VALUE

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

NOTES

For a complete description of this functionality, refer to: Burke, Edward, Michael Condict, David Mitchell, Franklin Reynolds, Peter Watkins, William Willcox, "RPC Design for Real-Time Mach", OSF Research Institute, Cambridge, MA, April 1994.

RELATED INFORMATION

Functions: **mach_rpc_trap**.

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

mach_rpc_signature

Structure— Signature for the Mach RPC paradigm

SYNOPSIS

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

DESCRIPTION

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

FIELDS

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

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

RELATED INFORMATION

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

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

thread_get_sched(2)

NAME

thread_get_sched - Get scheduling policy and parameters for a thread

SYNOPSIS

#include <mach/policy.h>

```
kern_return_t thread_get_sched(
        thread_act_t                  thr_act,
        policy_t                      *policy,
        sched_attr_t                  sched_attr,
        mach_msg_type_number_t        *sched_attrCnt);
```

DESCRIPTION

The **thread_get_sched** system call gets the scheduling policy and scheduling attributes for thread *thr_act*.

PARAMETERS

thr_act

The thread for which the scheduling policy and/or scheduling attributes are to be accessed.

policy

The scheduling policy that is currently scheduling *thr_act*. Scheduling policies are defined in **/usr/include/mach/policy.h**.

sched_attr

The scheduling attributes that are being used by *policy* to schedule *thr_act*. These scheduling attributes are defined in **/usr/include/mach/policy.h**.

sched_attrCnt

As an input parameter, *sched_attrCnt* specifies the size of the *sched_attr* structure, as a number of **int**'s. On completion of '**thread_get_sched**(),' *sched_attrCnt* specifies the size of the scheduling attribute structure returned in *sched_attr*, expressed as a number of **int**'s.

RETURN VALUE

KERN_INVALID_ARGUMENT
: Returned if *thr_act* is not valid (either because the thread activation is not valid or because it is not currently bound to a thread shuttle) or if *sched_attrCnt* is not large enough to accommodate a scheduling attribute structure for policy *policy*.

KERN_FAILURE
: Returned if the call to *policy* to access the thread's scheduling attributes fails.

Otherwise KERN_SUCCESS is returned.

NOTES

'**thread_get_sched**()' has been added to the kernel as part of the MK Scheduling Framework.

RELATED INFORMATION

thread_set_sched(2)

thread_set_sched(2)

NAME

thread_set_sched - Set scheduling policy and parameters for a thread

SYNOPSIS

#include <mach/policy.h>

kern_return_t **thread_set_sched**(

thread_act_t	*thr_act*,
policy_t	*policy*,
sched_attr_t	*sched_attr*,
mach_msg_type_number_t	*sched_attrCnt*);

DESCRIPTION

The **thread_set_sched** system call sets the scheduling attributes for thread *thr_act*. It can specify the scheduling policy to be used with the *policy* parameter, and the scheduling attributes for that policy with the *sched_attr* and *sched_attrCnt* parameters.

PARAMETERS

thr_act

The thread for which the scheduling policy and/or scheduling attributes are to be manipulated.

policy

The scheduling policy that should be used to schedule *thr_act*. Scheduling policies are defined in **/usr/include/mach/policy.h**.

sched_attr

The scheduling attributes that should be used by *policy* to schedule *thr_act*. These scheduling attributes are defined in **/usr/include/mach/policy.h**.

sched_attrCnt

The size of the scheduling attribute structure, expressed as a number of **int**'s.

RETURN VALUE

KERN_INVALID_ARGUMENT
: Returned if *thr_act* is not valid (either because the thread activation is not valid or because it is not currently bound to a thread shuttle) or if *sched_attrCnt* is not the correct size for a scheduling attribute structure for policy *policy*.

KERN_INVALID_POLICY
: Specified *policy* is not valid.

KERN_FAILURE
: Returned if the thread cannot be attached to policy *policy*, the assignment of the *sched_attr* attributes to the thread fails, or if the thread cannot be dispatched at the completion of the scheduling attribute modification.

Otherwise KERN_SUCCESS is returned.

NOTES

'**thread_set_sched**()' is intended to replace '**thread_set_policy**()' with the addition of the MK Scheduling Framework to the kernel.

RELATED INFORMATION

thread_get_sched(2), **thread_set_policy**(2)

Event Trace Analysis Package

etap_config(1)

NAME

etap_config - display ETAP configuration information

SYNOPSIS

etap_config

DESCRIPTION

The **etap_config** command writes to the standard output the current system's ETAP configuration. This information includes the number of entries in the event and subsystem tables, the size (in bytes) of the kernel's cumulative buffer, the size (in bytes) of a cumulative buffer entry, the maximum number of entries that the buffer can contain, and the current interval width (in microseconds). It also includes the size of the kernel's monitored buffer(s) (in bytes), the size of a monitored buffer entry (in bytes), the maximum number of entries the buffer can contain, and the number of monitored buffers. If either mode of tracing (cumulative or monitored) is not configured, the **etap_config** utility will print "NOT CONFIGURED" under the mode info heading.

EXIT VALUE

If the **etap_info**(2) system call returns failure, or if ETAP is not configured into the OSF/1 server, **etap_config** prints an error message and exits with a value of 1. Otherwise, it prints the configuration info and exits with a value of 0.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: CaraDonna, Joseph, "The Event Trace Analysis Package: ETAP User's Guide", OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

etap_status(1), **etap_trace**(1), **etap_md**(1), **etap_mscan**(1), **etap_probe**(2), **etap_trace_thread**(2)

etap_md(1)

NAME

etap_md - ETAP daemon: monitor and collect ETAP information

SYNOPSIS

etap_md [-**o** file] [-**t**] [-**p**N:NNN]

FLAGS

-o file
Use file as the base filename for the output file. The default name is /tmp/etap_mdata.log.

-t
Enable daemon thread event tracing

-pN:NNN
The user may specify the policy and priority via -p options. Policy can be [1-TS,2-RR,4-FIFO]. Priority can be [0-125]. The etap_md daemon spawns one thread per monitored buffer. Each spawned thread has the same policy and priority like its parents.

DESCRIPTION

When ETAP thread event or lock event monitoring is configured in the system, the kernel records traced data in circular, per-processor monitored buffers. **etap_md** is a multi-threaded daemon that monitors the kernel's monitored buffer(s) and writes the recorded data to disk as event activity persists.

The user may specify a base filename via the **-o** option; otherwise the base /tmp/etap_mdata.log. As mentioned previously, each processor has its own monitored buffer. Likewise, the **etap_md** daemon creates a separate file for each buffer. The file name is derived from the base name, where the processor number is appended to the end of the file name prior to the extension. For instance, if the base file name is etap_mdata.log and there are four active processors, the output files would be: etap_mdata0.log, etap_mdata1.log, etap_mdata2.log and etap_mdata3.log. All monitored buffer files are written in the ETAP Mdata file format.

The **etap_md** daemon spawns one thread per monitored buffer. Each thread writes the contents of its monitored buffer to disk as events occur, collecting data in real-time. If a buffer is not changing, its associated **etap_md** thread sleeps and periodically wakes-up to check the buffer. If the contents of the buffer have changed, the thread continues its process of writing the data to disk until buffer activity ceases.

Because the monitored buffers are circular, new event entries eventually overwrite old ones. Data is lost when kernel event trace activity occurs faster than the **etap_md** daemon can write them out to disk. For this reason, the **etap_md** daemon tracks data loss and periodically writes the current status out to disk. The default file name of the lost data log is /tmp/etap_mdata.lost. However, if the **-o** option is selected, the name of the lost data file is derived by appending a .lost extension to the specified base file name. Lost data files are written in the ETAP Mlost file format.

To terminate a trace and ensure that all data is written out to disk, send one of the following signals to the **etap_md** process: SIGINT, SIGHUP, SIGQUIT, or SIGTERM.

Once monitored event data is stored on disk, user-level utilities (i.e., **etap_mscan** and **etap_mstat**) may view and analyze the data to produce reports.

EXIT_VALUE

If **etap_md** is unable to open its log or data loss files, or if it is unable to allocate a mach port to sleep on and map the ETAP trace buffer into its address space, it prints a message and exits with a value of 1. Otherwise, when terminated by one of the abovementioned signals, it exits with a value of 0.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: CaraDonna, Joseph, “The Event Trace Analysis Package: ETAP User’s Guide”, OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

etap_status(1), **etap_trace**(1), **etap_config**(1), **etap_mscan**(1), **etap_probe**(2), **etap_trace_thread**(2)

etap_mscan(1)

NAME

etap_mscan - ETAP analysis utility

SYNOPSIS

etap_mscan [**-i** file] [**-v**] [**-l**]

FLAGS

-i file   Use file as the base name for validity and trace data files. The default is /tmp/etap_mdata.log.

-v   Generate a validity report

-l   Display event information on a single line.

DESCRIPTION

The **etap_mscan** is a sample analysis program which scans and displays the contents of the data and validity files created by the ETAP daemon (**etap_md**). This program simply opens the files, reads the headers and prints what it sees (somewhat formatted) to standard out. From this simple program, more complex tools can be derived to manage and give meaning to the traced data.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: Cara-Donna, Joseph, "The Event Trace Analysis Package: ETAP User's Guide", OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

etap_status(1), **etap_trace**(1), **etap_md**(1), **etap_config**(1), **etap_probe**(2), **etap_trace_thread**(2)

etap_resize(1)

NAME

etap_resize - dynamically change the number of entries in the ETAP event buffer

SYNOPSIS

etap_resize [num]

FLAGS

num The number of entries to be used in the ETAP event buffer

DESCRIPTION

etap_resize is used to dynamically change the number of event buffer entries in the system.

EXIT_VALUE

If **etap_resize** is run with no arguments, or num cannot be interpreted as a number, **etap_resize** prints a usage message and exits with a value of 1. If the system call to change the number of entries fails, **etap_resize** prints a message corresponding to the error code and exits with a value of 1. Otherwise, **etap_resize** exits with a value of 0.

RELATED INFORMATION

etap_status(1), **etap_trace**(1), **etap_config**(1), **etap_mscan**(1), **etap_probe**(2), **etap_trace_thread**(2)

etap_status(1)

NAME

etap_status - Display the contents of the kernel's ETAP event and subsystem tables

SYNOPSIS

etap_status [-**e**] [-**s**]

FLAGS

-e Display contents of event table (default)

-s Display contents of subsystem table

DESCRIPTION

The **etap_status** command writes to the standard output the contents of the ETAP event table and/or the contents of the ETAP subsystem table.

The ETAP event table contains a list of configured kernel events (both text and numerical representations) along with the current trace status for each. The trace status of an event may be either on or off in any of the following trace modes: cumulative hold, cumulative wait, monitored hold, monitored wait.

The subsystem table contains a list of configured ETAP subsystems, in both text and numerical representations.

EXIT VALUE

If ETAP is not configured into the kernel and OSF/1 server or the number of event table entries is zero, **etap_status** prints an error message to stderr and exits with a value of 1. Otherwise it prints the contents of the desired table and exits with a value of zero.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: Cara-Donna, Joseph, "The Event Trace Analysis Package: ETAP User's Guide", OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

etap_trace(1), **etap_config**(1), **etap_md**(1), **etap_mscan**(1), **etap_probe**(2), **etap_trace_thread**(2)

etap_trace(1)

NAME

etap_trace - Control modes of ETAP event tracing

SYNOPSIS

etap_trace [-**m**] [-**c**] [-**R**] [-**h**] [-**w**] [-**i**#] [-**d**] [-**e** type] [-**s** subsystem] [-**a**]

FLAGS

-m Monitored trace (default)

-c Cumulative trace

-R ETAP reset

-h Trace hold time - duration (default)

-w Trace wait time - contention

-i# Redefine the cumulative buffer interval width, where # is in microseconds. This flag can only be used when ETAP is being reset (via -**R**).

-d Disable tracing on the specified objects. If this flag is not used, tracing on specified objects is enabled.

-e type Enable (or if **-d** is used, disable) tracing on an event type. An event type can be a lock or a kernel probe.

-s subsystem
Enable (or if **-d** is used, disable) tracing on a subsystem.

-a Enable (or if -d is used, disable) tracing on all events.

DESCRIPTION

The **etap_trace** utility is used to enable and disable various modes of ETAP event tracing.

If a mode is not selected on the command line, **etap_trace** selects the monitored trace mode. Note: the trace modes available depend on the configuration of the kernel. If the reset mode (-**R**) is selected, all event tracing is disabled and the cumulative buffer is cleared. If the cumulative buffer is not configured in the kernel, then only event tracing affected.

Contention and duration operations are typically associated with the timing of lock events. When tracing general events, such as *p_thread_life*, either trace operation (hold or wait) may be used. If a type is not selected at the command line, **etap_trace** defaults to trace hold time.

When using the type parameter to redefine the cumulative buffer interval width, the **-i** switch is followed by a number specifying the width in microseconds. The cumulative buffer interval width can only be changed when ETAP is being reset (via the **-R** switch). This restriction prevents the collected distribution interval data from becoming distorted.

The **-d** switch is used to disable tracing on the specified objects. If **-d** is not selected at the command line, **etap_trace** enables tracing on the specified objects.

If either event type or subsystem is selected, a text name of the object is expected to follow. **etap_trace** accepts multiple instances of **-e** and **-s** on the command line.

EXAMPLES

To demonstrate the flexibility of the etap_trace utility, several examples of its usage are supplied below. Note: Defaults are used where applicable.

Enable cumulative hold tracing for the vm_map and ipc_object lock events:

```
etap_trace -c -e vm_map -e ipc_object
```

Disable all cumulative lock tracing without clearing the cumulative buffer:

```
etap_trace -c -d -h -w -a
```

Enable monitored tracing of the p_thread_life probe:

```
etap_trace -e p_thread_life
```

Enable monitored wait tracing for all kernel events:

```
etap_trace -w -a
```

Disable all monitored lock event tracing for the virtual memory subsystem:

```
etap_trace -d -s vm
```

Disable all tracing, clear the cumulative buffer and set the cumulative buffer interval width to 120 microseconds:

```
etap_trace -R -i 120
```

EXIT VALUE

If subsystem or event names are invalid, or if **-i** is used without **-R**, **etap_trace** prints a usage message and exits with a value of 1. Otherwise it performs the desired function and exits with a value of 0.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: Cara-Donna, Joseph, “The Event Trace Analysis Package: ETAP User’s Guide”, OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

etap_status(1), **etap_config**(1), **etap_md**(1), **etap_mscan**(1), **etap_probe**(2), **etap_trace_thread**(2)

etap_new_probe(2)

NAME

etap_new_probe - add a new named ETAP probe to the ETAP event table

SYNOPSIS

#include <mach/etap.h>

kern_return_t **etap_new_probe**(

char	**name*,
int	*namlen*,
int	*trace_on*,
unsigned short	**id*);

PARAMETERS

name

The probe's name

namlen

The length of the probe's name. *namlen* should be less than EVENT_NAME_LENGTH.

trace_on

A boolean value indicating the initial tracing status of the new probe. A value of 1 indicates tracing is initially on for the new probe; a value of 0 indicates that tracing is initially off.

id

A pointer to an unsigned short where the new probe id will be stored.

DESCRIPTION

The **etap_new_probe** system call is used to dynamically add a probe to the ETAP event table. The **etap_md** daemon should not be running while this system call is active, since it remaps the event table, and could cause **etap_md** to terminate abnormally with a memory protection error.

RETURN VALUE

If *namlen* is greater than or equal to EVENT_NAME_LENGTH, EINVAL is returned. If *id* does not point at valid storage, EFAULT is returned. If *name* already belongs to another probe in the table, KERN_NAME_EXISTS is returned. If there are more than 4095 entries in the probe table, KERN_RESOURCE_SHORTAGE is returned. If the kernel or OSF/1 server is not configured with ETAP, KERN_FAILURE is returned. Otherwise KERN_SUCCESS is returned.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: Cara-Donna, Joseph, “The Event Trace Analysis Package: ETAP User’s Guide”, OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

etap_status(1), **etap_trace**(1), **etap_config**(1), **etap_mscan**(1), **etap_trace_thread**(2), **etap_probe**(2)

etap_probe(2)

NAME

etap_probe - user-level ETAP probe

SYNOPSIS

#include <mach/etap.h>

```
kern_return_t etap_probe(
        unsigned short        event_type,
        unsigned short        event_id,
        unsigned int          data_size,
        etap_data_t           *data);
```

PARAMETERS

event_type
: The user event number. The event number is either a number between 0 and 31 or a number returned by **etap_new_probe**(2).

event_id
: A user-defined value used to further identify an event.

data_size
: The size (in bytes) of the data being passed. The size is limited to ETAP_DATA_SIZE in etap.h

data
: A pointer to the data being passed in

DESCRIPTION

Application programmers may instrument their applications with user-level probes using the **etap_probe** system call. All event data passed in is recorded in the kernel's ETAP monitored buffer(s).

RETURN VALUE

If *data_size* is greater than ETAP_DATA_SIZE, **etap_probe** returns KERN_INVALID_ARGUMENT. If the event is not enabled or *event_type* is not either between 0 and 31 or a value returned by **etap_new_probe**, **etap_probe** returns KERN_NO_ACCESS. If ETAP event monitoring is not configured into the kernel, KERN_FAILURE is returned. Otherwise **etap_probe** returns KERN_SUCCESS.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: Cara-Donna, Joseph, "The Event Trace Analysis Package: ETAP User's Guide", OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

etap_status(1), **etap_trace**(1), **etap_config**(1), **etap_mscan**(1), **etap_new_probe**(2), **etap_trace_thread**(2)

etap_trace_event(2)

Function - Enables/disables kernel ETAP event probes.

LIBRARY

#include <mach/etap.h>

SYNOPSIS

kern_return_t **etap_trace_event**

(unsigned short	*mode,*
unsigned short	*type,*
boolean_t	*enable,*
unsigned int	*nevents,*
unsigned short	**event_list*);

DESCRIPTION

The **etap_trace_event()** system call enables/disables ETAP trace modes for the specified list of events.

PARAMETERS

mode

[in scalar] Must be one of:

ETAP_RESET - reset: clears all trace modes for all events. Clears the cumulative buffer.

ETAP_CUMULATIVE - Cumulative trace mode

ETAP_MONITORED - Monitored trace mode

type

[in scalar] Must be one of:

ETAP_DURATION - measure operation duration

ETAP_CONTENTION - measure operation contention

or the logical OR of both

enable

[in scalar] Enable/disable switch where:

TRUE = enable

FALSE = disable

nevents

[in scalar] The number of events in the event list array

event_list

[in array pointer] Pointer to an array of ETAP events

RETURN VALUE

Only generic returns apply.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: Cara-Donna, Joseph, “The Event Trace Analysis Package: ETAP User’s Guide”, OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

Functions: **etap_probe**(2), **etap_trace_thread**(2).

etap_trace_thread(2)

NAME

etap_trace_thread - control tracing of ETAP probes

SYNOPSIS

#include <mach/etap.h>

kern_return_t **etap_trace_thread**(
mach_port_t *target_thread*,
boolean_t *active*);

PARAMETERS

target_thread

The port of the thread whose ETAP trace status will be specified

active

The boolean value stating whether the thread's ETAP trace status will be enabled or not. Passing TRUE will enable the thread's trace status and FALSE will deactivate it.

DESCRIPTION

The **etap_trace_thread** system call is used to specify the ETAP trace status of a thread.

RETURN VALUE

If *target_thread* does not correspond to a valid thread, **etap_trace_thread** returns KERN_INVALID_ARGUMENT. If ETAP event monitoring is not configured into the kernel KERN_FAILURE is returned. Otherwise KERN_SUCCESS is returned.

NOTES

For a complete description of the Event Trace Analysis Package, refer to: CaraDonna, Joseph, "The Event Trace Analysis Package: ETAP User's Guide", OSF Research Institute, Cambridge, MA, February 1, 1995.

RELATED INFORMATION

etap_status(1), **etap_trace**(1), **etap_config**(1), **etap_mscan**(1), **etap_new_probe**(2), **etap_probe**(2)

Clocks and Timers

clock_mgt(2)

NAME

clock_mgt - Clock management functions.

LIBRARY

Standard C Library (libc.a):

SYNOPSIS

#include <sys/time.h>

```
int clock_mgt(
        int     fn_code,
        void    *value);
```

PARAMETERS

fn_code

One of set to select function to be performed.

value

Datum to be read or written

DESCRIPTION

The **clock_mgt** function provides special functions for managing the system clock. The function to be performed is designated by *fn_code*, one of the following:

CLOCK_GET_FREQ

Get the current frequency of the real-time hardware clock device. The frequency is an int returned in **value*.

CLOCK_SET_FREQ

Set the frequency of the real-time hardware clock device to **value* (expressed in Hertz). This does not change anything in the hardware; rather it conditions the kernel clock service for the frequency at which the hardware clock crystal is assumed to be operating. The value is machine-dependent. For PC-AT platforms, the nominal value is 1193167.

CLOCK_GET_BB

Get the current time from the hardware battery-backed (CMOS) clock. The time is a struct timeval returned in **value*.

CLOCK_SET_BB

Set the time in the hardware battery-backed (CMOS) clock to **value* which should be a struct timeval. This clock is machine-dependent. For PC-AT platforms, time is kept only to 1-second resolution.

CLOCK_MAP

Map (read-only) the time value of the real-time hardware clock and related variables into an empty page in current address space. The data is contained within a struct mapped_timespec structure located at offset 0 within the mapped page. struct mapped_timespec is defined in mach/clock_types.h. This function is implemented with the kernel **clock_map_time** function.

NOTES

A process must have superuser privilege to set the frequency or the CMOS clock time.

RETURN VALUES

Upon successful completion, a value of 0 (zero) is returned. Otherwise, a value of -1 is returned, and errno is set to indicate the error.

ERRORS

EFAULT A parameter points to an invalid address.

EPERM The process's effective user ID does not have superuser privilege.

EINVAL Illegal value for parameter.

RELATED INFORMATION

gettimeofday(2), **settimeofday**(2)

ntp_adjtime(2)

NAME

ntp_adjtime - NTP-specified adjust time function.

LIBRARY

Standard C Library (libc.a):

SYNOPSIS

#include <sys/timex.h>

int **ntp_adjtime**(
    struct timex * *tptr*);

DESCRIPTION

This function supports the Network Time Protocol.

For more information, see Mills, D.L. “Unix kernel modifications for precision time synchronization,” Electrical Engineering Department Report 94-10-1, University of Delaware, October 1994. As of the publication of these release notes, this report is available on the web at:

http://www.eecis.udel.edu/~mills/reports.html

RELATED INFORMATION

gettimeofday(2), **ntp_gettime**(2), **settimeofday**(2)

ntp_gettime(2)

NAME

ntp_gettime - NTP-specified get time function.

LIBRARY

Standard C Library (libc.a):

SYNOPSIS

#include <sys/timex.h>

int **ntp_gettime**(
struct ntptimeval * **tptr*);

DESCRIPTION

This function supports the Network Time Protocol.

For more information, see Mills, D.L. "Unix kernel modifications for precision time synchronization," Electrical Engineering Department Report 94-10-1, University of Delaware, October 1994. As of the publication of these release notes, this report is available on the web at:

http://www.eecis.udel.edu/~mills/reports.html

RELATED INFORMATION

gettimeofday(2), **ntp_adjtime**(2), **settimeofday**(2)

clock_gettime(3)

NAME

clock_gettime - Gets the current value of a system-wide clock

LIBRARY

Standard C Library (libc.a)

SYNOPSIS

```
#include <sys/timers.h>

int clock_gettime(
        clockid_t                    clock_id,
        struct timespec              *tp);
```

PARAMETERS

clock_id
: Identifies a system-wide clock.

tp
: Points to a timespec structure space where the current value of the system-wide clock is stored.

DESCRIPTION

The **clock_gettime**() function sets the current value of the clock specified by *clock_id* into the location pointed to by the *tp* parameter.

The *clock_id* parameter is given as a symbolic constant name, as defined in the **sys/timers.h** header file. It must be one of the following:

CLOCK_REALTIME
: The normal time-of-day clock. This is the elapsed time since the Epoch. The Epoch is referenced to 00:00:00 UTC (Coordinated Universal Time) January 1, 1970.

CLOCK_SYSTIME
: This is the local machine time base. It is the elapsed time since system startup or some arbitrary event prior to that. Its rate is not adjusted after system startup is complete, and it is not affected by “leap seconds” or other artifacts of UTC.

The **timespec** structure is defined in the **sys/timers.h** header file and has the following members:

unsigned long tv_sec
Seconds.

long tv_nsec
Fraction of a second in units of nanoseconds.

The time interval expressed by the members of this structure is ((**tv_sec** * 10^9) + (**tv_nsec**)) nanoseconds.

NOTES

The **clock_gettime**() function is part of the POSIX 1003.1 Standard, 1996 Edition.

The time value returned from the **CLOCK_REALTIME** system clock is the same as from **gettimeofday**(). **clock_gettime**() is much faster than **gettimeofday**() on some platforms and is the recommended call for all user-mode applications. The overhead of getting the **CLOCK_SYSTIME** system clock is as low as or lower than **CLOCK_REALTIME**.

RETURN VALUES

Upon successful completion, the **clock_gettime**() function returns a value of 0 (zero). Otherwise, **clock_gettime**() returns a value of -1, and **errno** is set to indicate the error.

ERRORS

If any of the following conditions occurs, the **clock_gettime**() function sets **errno** to the value that corresponds to the condition.

EINVALThe *clock_id* parameter does not specify a known system-wide clock.

EIO An error occurred when the system-wide clock specified by the *clock_id* parameter was accessed.

RELATED INFORMATION

Functions: **timer_gettime**(3), **clock_settime**(3), **time**(3).

clock_settime(3)

NAME

clock_settime - Sets the value of a system-wide clock

LIBRARY

Standard C Library (libc.a)

SYNOPSIS

#include <sys/timers.h>

int **clock_settime**(
    clockid_t clock_id,
    struct timespec *tp);

PARAMETERS

clock_id Specifies a system-wide clock.

tp Points to the location where the value of the time to be set into the clock specified by the *clock_id* parameter is stored.

DESCRIPTION

The **clock_settime**() function sets a time value into the system-wide clock specified by the *clock_id* parameter.

The *clock_id* parameter is given as a symbolic constant name, as defined in the **sys/timers.h** header file. It must be one of the following:

CLOCK_REALTIME
The normal time-of-day clock. This is the elapsed time since the Epoch. The Epoch is referenced to 00:00:00 UTC (Coordinated Universal Time) January 1, 1970.

The value of time set by this function is obtained from a type timespec structure pointed to by the *tp* parameter. The **timespec** structure represents an amount of time. It is defined in the **sys/timers.h** header file and has the following members:

unsigned long tv_sec
Seconds.

long tv_nsec
Fraction of a second in units of nanoseconds.

The time interval expressed by the members of this structure is ((**tv_sec** * 10^9) + (**tv_nsec**)) nanoseconds.

NOTES

The **CLOCK_SYSTIME** system clock cannot be set.

The **clock_settime**() function is part of the POSIX 1003.1 Standard, 1996 Edition.

RETURN VALUES

Upon successful completion, the **clock_settime**() function returns a value of 0 (zero). Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

ERRORS

If any of the following conditions occurs, the **clock_settime**() function sets **errno** to the value that corresponds to the condition.

EINVAL The clock_id parameter does not specify a known system-wide clock; or the information pointed to by the tp parameter is outside the permissible range for the clock specified by the clock_id parameter; or a nanosecond value less than 0 (zero) or greater than or equal to 1000 million is specified by the information pointed to by the tp parameter.

EIO An error occurred while accessing the clock specified by the clock_id parameter.

EPERM
The requesting process does not have the appropriate privilege to set the clock specified by the clock_id parameter.

RELATED INFORMATION

Functions: **clock_gettime**(3), **timer_gettime**(3), **time**(3).

timer_create(3)

NAME

timer_create - Allocates a per-process timer

LIBRARY

Standard C Library (libc.a)

SYNOPSIS

#include <signal.h>

#include <sys/timers.h>

```
timer_t timer_create(
        clockid_t           clock_id,
        struct sigevent     *evp,
        timer_t             *timerid);
```

PARAMETERS

clock_id Specifies the system-wide clock to be used as a per-process time base for the new timer.

evp Specifies the mechanism by which a process is to be notified when the per-process timer times out.

timerid A timer_t value that identifies the new per-process timer.

DESCRIPTION

The **timer_create**() function is used to allocate a per-process timer using a specified system-wide clock as its time base. The **timer_create**() function returns a unique timer ID, *timerid*, which is used to identify the timer in per-process timer requests.

OSF/1 allows each process to allocate one per-process timer. The **CLOCK_REALTIME** system clock must be specified by the *clock_id* parameter. The **DELIVERY_SIGNALS** notification mechanism must be specified by the **sigev_notify** member of the *evp* parameter. With this mechanism, the system sends a **SIGALRM** signal to the process whenever the timer expires.

NOTES

Per-process timers are not inherited by a child process across **fork**() or **exec...**() functions.

The **timer_create**() function is part of the POSIX 1003.1 Standard, 1996 Edition.

RETURN VALUES

Upon successful completion, the **timer_create**() function returns a value of 0 (zero). Otherwise, **timer_create**() returns a value of -1, and **errno** is set to indicate the error.

ERRORS

If any of the following conditions occurs, the **timer_create**() function sets **errno** to the value that corresponds to the condition.

EAGAIN
The calling process has already allocated all available timers.

EINVAL
The *clock_id* parameter or the **sigev_notify** member of the *evp* parameter is invalid.

RELATED INFORMATION

Functions: **exec**(2), **fork**(2), **clock_gettime**(3), **timer_gettime**(3), **timer_settime**(3), **timer_delete**(3), **clock_settime**(3).

timer_delete(3)

NAME

timer_delete - Frees a per-process timer

LIBRARY

Standard C Library (libc.a)

SYNOPSIS

#include <sys/timers.h>

int **timer_delete**(
timer_t *timer_id*);

PARAMETERS

timer_id
Specifies the per-process timer to deallocate.

DESCRIPTION

The **timer_delete**() function is used to free a previously allocated per-process timer (previously returned by the **timer_create**() function). Any pending per-process timer event generated by the timer specified by the *timer_id* parameter is canceled when this function successfully executes.

NOTES

The **timer_delete**() function is part of the POSIX 1003.1 Standard, 1996 Edition.

RETURN VALUES

Upon successful completion, a value of 0 (zero) is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

ERRORS

If the following condition occurs, the **timer_delete**() function sets **errno** to the value that corresponds to the condition.

EINVAL
The timerid parameter does not specify an allocated per-process timer.

RELATED INFORMATION

Functions: **timer_gettime**(3), **timer_create**(3), **timer_settime**(3).

timer_gettime(3)

NAME

timer_gettime - Gets the value of a timer

LIBRARY

Standard C Library (libc.a)

SYNOPSIS

#include <sys/timers.h>

int **timer_gettime**(
timer_t *timerid*,
struct itimerspec **tp*);

PARAMETERS

timerid Specifies the per-process timer whose value is to be retrieved.

tp Points to an itimerspec structure.

DESCRIPTION

The **timer_gettime**() function gets the current value of a per-process timer. The *timerid* parameter specifies the timer. The value of *timerid* must correspond to a timer_t value returned by the **timer_create**() call. The *tp* parameter points to a type **itimerspec** structure, which has an **it_value** member that specifies the amount of time remaining before the timer expires. The **itimerspec** structure is defined in the **sys/timers.h** header file.

Actual resolution of a timer is determined by the basic system hardware clock period.

NOTES

The **timer_gettime**() function is part of the POSIX 1003.1 Standard, 1996 Edition.

RETURN VALUES

Upon successful completion, the **timer_gettime**() function returns the value 0 (zero). Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

ERRORS

If any of the following conditions occurs, the **timer_gettime**() function sets **errno** to the value that corresponds to the condition.

EINVAL The *timerid* parameter does not specify a known timer.

EIO An error occurred while accessing the clock device.

RELATED INFORMATION

Functions: **alarm**(3), **clock_gettime**(3), **timer_create**(3).

timer_settime(3)

NAME

timer_settime - Establishes time-out intervals of a per- process timer

LIBRARY

Standard C Library (libc.a)

SYNOPSIS

#include <sys/timers.h>

int **timer_settime**(
    timer_t *timerid*,
    int *flags*,
    struct itimerspec **val*,
    struct itimerspec **oval*);

PARAMETERS

timerid Specifies the per-process timer to access.

flags Flags (none presently implemented.)

val Points to a type **itimerspec** structure that contains the values of the initial and offset time-out intervals.

oval Points to a type **itimerspec** structure where the current value of the timer time-out interval and the time-to-go value are to be stored.

DESCRIPTION

The **timer_settime**() function sets a timer to the specified value *val*, returning the previous value of the timer if *ovalue* is nonzero.

A timer value is defined by the **itimerspec** structure:

```
struct itimerspec {
   struct  timespec it_interval;
   struct  timespec it_value;
};
```

If the **it_value** field is nonzero, it indicates the time to the next timer expiration. If the **it_interval** field is nonzero, it specifies a value to be used in reloading

it_value when the timer expires. Setting **it_value** to 0 (zero) disables a timer. Setting **it_interval** to 0 (zero) causes a timer to be disabled after its next expiration, assuming **it_value** is nonzero.

NOTES

Time values smaller than the resolution of the system clock are rounded up to this resolution.

The **timer_settime**() function is part of the POSIX 1003.1 Standard, 1996 Edition.

RETURN VALUES

Upon successful completion, the **timer_settime**() function returns a value of 0 (zero). Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

ERRORS

If the following condition occurs, the **timer_settime**() function sets **errno** to the value that corresponds to the condition.

EINVAL The *timerid* parameter does not specify an allocated per-process timer, or the *val* parameter points to a nanosecond value less than zero or greater than or equal to 1000 million.

RELATED INFORMATION

Functions: **alarm**(3), **clock_gettime**(3), **timer_gettime**(3), **timer_create**(3).

time_stamp_counter(3)

NAME

time_stamp_counter - Functions implemented via macros for inline use of the Pentium Time Stamp Counter (TSC).

LIBRARY

None.

SYNOPSIS

#include <mach/i386/tsc.h>

void **rd_tsc_nsec**(
    tvalspec_t **tp*);

void **rd_tsc_usec**(
    time_value_t **tp*);

void **rd_tsc**(
    quad_t **tp*);

void **tsc_timeofday_ns**(
    tvalspec_t **tp*);

void **tsc_timeofday_us**(
    time_value_t **tp*);

boolean_t **tsc_present**(
    void);

PARAMETERS

tp Pointer to datum of the indicated type.

DESCRIPTION

These functions are implemented as macros in the **mach/i386/tsc.h** header file. The macros expand to inline **asm** calls. Except for **tsc_present**, these functions are only available on Pentium-based MK7.3 or later systems, and their behavior is undefined otherwise. The **tsc_present** function may be used in MK7.3 on any hardware platform to determine if the other functions are available. For normal application use, the POSIX compliant function **clock_gettime**(3) is recommended.

rd_tsc_nsec
Read the TSC, transform that counter value into a time based on the

REALTIME_CLOCK, and represent the resultant value in **tvalspec_t** (nanosecond precision) form. The provides the same time value as the kernel **REALTIME_CLOCK** service.

rd_tsc_usec
Read the TSC, transform that counter value into a time based on the **REALTIME_CLOCK,** and represent the resultant value in **time_value_t** (microsecond precision) form. This provides the same time value as the kernel **REALTIME_CLOCK** service as converted to microsecond precision.

rd_tsc Read the TSC and store the quadword value without conversion.

tsc_timeofday_ns
Read the TSC, transform that counter value into a time, and present that time as a **tvalspec_t** (nanosecond precision) representing UTC time of day. This is the same time value as that returned by **clock_gettime**(3).

tsc_timeofday_us
Read the TSC, transform that counter value into a time, and present that time as a **time_value_t** (microsecond precision) representing the UTC time of day. This is the same time value as that returned by **gettimeofday**(2).

tsc_present
Returns a boolean value which is true if the TSC functions above are available for use.

RETURN VALUES

The **tsc_present** function returns a boolean value.

ERRORS

No errors are defined. If the TSC is not present, the effect of these functions (other than **tsc_present**) is undefined.

RELATED INFORMATION

Functions: **clock_gettime**(3), **time**(3).

uclock(8)

NAME

uclock - Utility Clock operations

SYNOPSIS

/sbin/uclock [option [parameter]]

DESCRIPTION

This command performs various operations on system clocks, including the battery-backed (CMOS) clock and the system realtime clock. The operation is selected from among the options below. (p) means requires root uid or sysadmin privs.

no arg   report bb clock, system time, and difference.

attr   report rt clock frequency

setst   set system time from bb clock. (p)

setbb   set bb clock from system time. (p)

fix   fix system time by adjustment toward bb clock time. (p)

slide N   slide (adjust) system time by +/- N microseconds. (p)

cal N   calibrate system time with bb clock over N seconds. (p)

check N check system time against bb clock over N seconds.

start {bb|ntp}
    set rt clock freq at system startup, track ntp or bb clock (p)

adjust N make rt clock run faster (+N) or slower (-N) by N ppm. (p)

freq N   set rt clock assumed frequency to N. (p)

This utility provides an interface to the kernel support for calibrating the frequency of the realtime clock. All system time intervals or values are derived from this clock, so calibration at this level improves the accuracy of all times and prevents divergence of time values obtained from different syscalls.

On machines running NTP, **uclock** is run at system startup with the 'start' option and uses information left by xntpd to set the clock frequency so as to remove as much drift as possible. It saves the new clock frequency in **/etc/sclock.freq** and will continue to adjust it at each system startup.

The frequency may be manually recalibrated by the command '**uclock** start ntp', but it's only worth doing this if the **ntp.drift** file has been updated by NTP since the system started. Also, it is known that certain clock clients (LTS in particular) will be adversely affected if the clock frequency is changed significantly during normal operation.

It is possible to use frequency calibration on machines not running NTP also. In this case, the file init.d/settime should contain '**uclock** start bb'. Then, observe how well the system keeps time by comparing system time to some external standard over a sufficiently long time (a day or more) to compute the drift in parts-per-million. For example, if the clock loses 10 seconds in a day:

drift.ppm = 1000000 * (10 / 86400) = 115

This means we want the clock to run faster by about 115 ppm. The following command will do that:

/sbin/uclock adjust 115

If the clock were gaining, the argument to **uclock** would be negative.

The 'adjust' option computes and sets a new frequency which will be used until manually changed. Two or three cycles of adjustment should be sufficient to get the frequency calibrated as accurately as possible. Typical i386 platforms exhibit a stability of +/- 3 ppm or less over the course of a day. Hence, a long term drift of 1 second/day or more should be correctable. Because the basic clock frequency is on the order of 1 MHz, the lower limit of correctability is about 1 ppm.

Some **uclock** options allow for calibrating the rt clock to the bb clock in the absence of a better external standard. However, typical bb clocks are only accurate to 20 ppm or so. The 'check' option can be used to compare the rates of the system and bb clock precisely over a short period of time. The 'cal' option does the same comparison and then sets the rt clock frequency to match the rate of the bb clock.

The 'slide' option uses the traditional **adjtime**() syscall, while the 'adjust' option adjusts the low-level kernel rt clock rate.

FILES

/etc/sclock.freq

Records the last clock frequency determined from the NTP drift rate or from the 'adjust' option. Used to set the rt clock frequency at system startup.

/var/adm/ntp/ntp.drift

Written by xntpd periodically when synchronized to an external standard; represents the long-term estimated drift of the system clock in ppm. Used by **uclock** at system startup to compute a new estimate for the rt clock frequency.

EXIT VALUES

Returns 0 if the command was executed with no errors. The following error values may be returned:

NO_ROOT	"NOT super-user"
BADARG	"unknown arg"
FFLFAIL	"Unable to write clock frequency file: "
GETFRQ	"Failed to get current clock freq, rv = N"
SETFRQ	"Failed to set clock freq, rv = N"
NEWFREQ	"Clock freq set to N, previous freq was M"
NOBB	"get bbc time failed, rv = N"
BBJUMP	"bbc changed by N seconds!"
SUBFREQ	"Unable to get current clock freq, assuming N"
GETTFAIL	"GETTIME failed, rv = N"
SETTFAIL	"settimeofday failed, rv = N"
SETOK	"time set ok"
ADJFAIL	"adjtime failed, rv = N"
NOSETBB	"set bb clock failed, rv = N"
BBOK	"bb clock set ok."
MINADJ	"unable to adjust clock by less than N ppm."
XSADJ	"max adjustment is +/- 1000 ppm."
ADJTOOBIG	"adjustment too large"
AUTH	"need sysadmin authorization"

PRIV "insufficient privileges"

RELATED INFORMATION

gettimeofday(2), **settimeofday**(2), **clock_mgt**(2)

CORDS and Communication

prottbl(4)

NAME

prottbl - format of CORDS protocol table file

DESCRIPTION

The **prottbl** file is a CORDS configuration file which lists CORDS protocol id numbers.

The name and location of the protocol table file is specified in the graph file using the *prottbl*= line.

The usual location is **/mach_servers/prottbl**.

SEE ALSO

CORDS manual

rom(4)

NAME

rom - format of CORDS rom file

DESCRIPTION

A rom file is used to provide run-time parameters to its CORDS protocols and framework.

Each line in the rom file specifies the protocol or framework option which it configures, a keyword that specifies which parameter and the values(s) for the parameter.

The framework options are *scheduling*, which specifies thread priorities, and *clock*, which is used by the CORDS event subsystem. Valid options are:

scheduling <u>thread</u> <u>priority</u>

where <u>thread</u> is one of *input_thread_priority*, *event_thread_priority*, *clock_thread_priority*, or *driver_thread_priority*, and <u>priority</u> is integer priority value. (Note that this is a kernel priority, as opposed to any other notion of priority that might exist, such as POSIX thread priorities.)

clock granularity <u>time</u>

where <u>time</u> is the event granularity in milliseconds.

Other framework rom options are used to configure CORDS paths and allocators. See the CORDS manual for more details.

Protocol-specific rom options are documented along with the protocol.

A compile-time value for a rom option may be provided in the **graph.comp** file, using the keyword *romopt*. This is the usual way of providing options to CORDS protocols in the kernel.

SEE ALSO

dtime_mgmt(8), **gipc**(8), **ballsorter**(8)

RELATED INFORMATION

See the CORDS manual for additional details.

abcasttest(8)

NAME

abcasttest - send atomic broadcast packets for testing

SYNOPSIS

abcasttest

DESCRIPTION

The **abcasttest** utility runs symmetrically on a group of nodes. The master must be started first and become ready before the others. **Abcasttest** uses the atomicbcast test protocol to exercise the CORDS sequencer with test packets using a window of slot buffers for the packet states. The message sending service synchronizes with the receiving service and continuously sends out testing packets until terminated (by SIGINT). For every one thousand atomic broadcasts, **abcasttest** prints out the total transmission time.

In order to perform jitter analysis, the JITTER_ANALYSIS conditional compilation flag should be set to 1 in the source file atomicbcast.c. After recompiling the program, **abcasttest** will report individual packet time measurements recorded during every one thousand atomic broadcasts.

To see how all slaves perform without output for the master, set the SILENT_MASTER conditional compilation flag to 1 in the source file atomicbcast.c and recompile the program.

FILES

mach_services/cmds/cords_tests
: location of the source code

prottbl
: protocol table (location can be specified by the ROM file)

ROM file
: please refer to the document “The x-kernel-based Communication Subsystem for OSF Mach: changes to the x-kernel 3.2 API”, Appendix A for complete details.

RELATED INFORMATION

updiptest(8)

balldemo(8)

NAME

balldemo - real-time ball-sorting demo

SYNOPSIS

balldemo [options]

FLAGS

These options are the most likely to be used during ball sorting:

-a address
Least significant octet of IP address (in decimal format). (Required when running in single user mode. The more significant portion of the IP addresses of the machines running the demonstration must all be the same.)

-b mps Markers per second for marker demo (with **-r**). Default rate is 2.

-p Node is attached to the ball-sorting device. Exactly one node is so designated in order to run the ballsorter demo, and it is referred to as the ball-handling node.

-r Run the marker demo. If any node in the group is running the marker demo, all the nodes must do so.

The following flags are obscure and were primarily used for development and tuning.

-c color Threshold value for red vs. black determination. Default is 3400. All pool nodes should use the same value.

-d pool_delay
Amount of delay which pool nodes allow for when sending a ‘flip’ message. Default is 10ms. All pool nodes should use the same value.

-s sensor_delay
Approximate minimum time delay between the first and second sensors. Default is 0.140 sec. Meaningful only to ball-handling node.

-t text Run marker demo in text/debugging mode (with **-r**).

DESCRIPTION

The balldemo program is the driver for a demonstration of real-time fault tolerant communication and control work done at the Open Group Research Institute. See http://www.opengroup.org/RI/PubProjPgs/CORDS.htm

The demonstration exercises two interesting features of this work: the ballsorter, in which a physical apparatus sorts red and black golf balls using a distributed control mechanism, and the ring, which shows how a consistent view of membership is maintained by all the group members.

The ballsorter:

In the ballsorter portion of the demo, red and black golf balls drop through a tube. Two sensors on the tube relay color and velocity information to a computer. This computer also drives a controller which flips a flipper at the bottom of the tube, directing the ball either right or left. As the flippers have a non-zero reset time and there may be more than one ball in the system simultaneously, it is important to move the flipper just before the ball reaches it and to reset it immediately after is has passed.

The computer directly connected to the physical apparatus is called the ball-handling node. The other computers are a pool of computational resources to which the ball-handling node sends raw sensor data and from which it receives high-level control information. Together, all these machines form a group, which uses the GIPC service for assured communication among all the members.

As long as at least one pool node is in the communication group with the ball-handling node through the duration of the ball dropping through the tube, the correct operation of the ballsorter will continue despite leaves or joins from the pool.

The ball sorting task is an excellent model of more complex problems.

A simplified description of the operation of the system follows.

On the ball-handling node, there is a pool of threads which are understood as one pair per ball in the system (for a maximum of 5 live balls). One member of the pair is responsible for sensing color and velocity, the other controls the flipper. These threads are scheduled by dispatch threads, which poll the sensor and network inputs. Another set of threads (of less priority) perform the marker demo operations.

1. On the ball-handling node, a thread continually polling the first sensor detects a ball and awakens a thread.

2. This thread sleeps for sensor_delay, then begins polling the second sensor. This delay is somewhat arbitrary, since the system has no information about the initial velocity and it is possible to “drop” the ball with sufficient force that the

second sensor does not awaken in time. If the second sensor does not detect a ball soon after the thread wakes up, the ball is given up for lost and system continues to function.

3. The thread sends the color and velocity data, along with a distributed timestamp to the pool, using GIPC communication mechanism which guarantees delivery to all live nodes. The thread goes to sleep.

4. A thread at each pool node receives the data.

5. Using a distributed clock synchronized via LTS, each pool node determines when (according to its local clock) and for which color the flipper should move.

6. The thread sleeps until pool_delay before that time. This delay allows for the wakeup resolution (< 5ms), the synchronization bound (< 2 * 1.25 ms), time for the reply (< 1ms) and miscellaneous delays. Values as low as 6 or 7 ms for the pool_delay appear to be quite acceptable.

7. The thread wakes up and uses a CORDS unreliable point-to-point protocol layered over UDP to send a control message back to the ball-handling node telling it which flipper to move. Each pool node that was in the group when the sensor data was posted and is still in the group will send a control message.

8. The first message which arrives at the ball-handling node is received by thread which wakes up the appropriate flipper-controlling thread. Subsequent messages referring to this ball are ignored. (The point to point protocol is used due to limitations of the current GIPC service; in principle, all the messages could be dropped and the ball sorter would fail.)

9. The thread controlling the flips the flipper, sleeps briefly (allowing the ball to pass the flipper), then resets it.

The ballsorter application extensively uses real time features of the OSF MK, such as real-time threads and synchronizers.

The marker demo:

The marker demo portion of the system is a modified version of the sample GIPC application. It demonstrates how the system maintains a consistent view of group membership and solves the consensus problem in the presence of various node or link failures.

The marker demonstration, also referred to as the ring demonstration, is described in **demo_client**(8).

The combined demo:

The combined demo is a combination of the two demos. The contents of the reliable broadcast messages have a type field that indicates whether they are the marker demo marker or the ballsorter sensor data. Messages are then forwarded to either a low priority marker demo thread or a high priority ballsorter thread. Similarly, the threads that wake up to post the marker have lower priority than threads which wakeup to poll the sensor or control the flipper. The most significant modifications were those needed to arrange appropriate thread priorities for the interacting demos.

Running the demos:

The demo may be run in single or multi-user mode.

If running in single user mode, run bcheckrc (termcap entries) and set the terminal type (for curses).

To run only the marker demo: on any node, start the GIPC server in the background, then run the demo application. This first node must finish initializing itself (several seconds) before other nodes may join the group. In the case of the marker demo, the marker appears shortly after initialization is complete. The remaining nodes may join simultaneously.

In the case of the marker demo, no node is considered protected. Any combination of nodes may be reset at any time.

The visual effect of the marker demo is best seen when the participating nodes (or their monitors) are arranged in order of increasing IP address.

The ballsorter demo is run similarly to the marker demo, but has additional requirements.

The ball-handling node is the one to which the ballsorter apparatus is connected. In principle, this can be any node of the group. However, the configuration in which the demo has been most thoroughly exercised ensures that the ball-handling node will always be the GIPC master node. In this case, the node attached to the ball apparatus must be the node with the highest IP address and it is on this ball-handling node that the application must start first.

In addition to the GIPC server, the LTS manager (dtime_mgmt) must also be running in the background. The pool nodes rely on LTS to control the apparatus, so the LTS master node must remain available. The LTS service can be configured such that the ball-handling node is the LTS master or the pool node running the master can be designated as a protected resource. The former is preferable, since, if the LTS master clock is located on the ball-handling node, the difference between the clock value at the ball-handling node and all of the pool nodes is at most the synchronization error bound. Whereas, if the LTS master clock is located on one of the pool nodes, the total error between the ball-handling node and any pool node can be twice the synchronization error bound, since each can vary by the amount of the synchronization bound.

The rom file must specify the destination address for the point to point (ptp) protocol. This address will be the address of the protected node. The rom file line is:

ptp server 130.105.2.82 #example

The rom file also describes the real-time scheduling parameters for the ballsorter system.

EXAMPLES

Consider three nodes with IP addresses 130.105.2.21, 130.105.2.30, and 130.105.2.82.

1. Run just the marker demo, moving the marker 9 times per second.

on 130.105.2.21 : balldemo -a 21 -r -b 9

on 130.105.2.30 : balldemo -a 30 -r -b 9

on 130.105.2.82 : balldemo -a 82 -r -b 9

2. Run only the ball sorter. The ballsorter apparatus is attached to node 130.105.2.82.

on 130.105.2.21 : balldemo -a 21

on 130.105.2.30 : balldemo -a 30

on 130.105.2.82 : balldemo -a 82 -p

3. Run both the marker demo and ball sorter. The ballsorter apparatus is attached to node 130.105.2.82. The apparatus is brightly lit, so modify the threshold sensor value for the red/black decision.

on 130.105.2.21 : balldemo -a 21 -r -c 3600

on 130.105.2.30 : balldemo -a 30 -r -c 3600

on 130.105.2.82 : balldemo -a 82 -r -p

FILES

rom(4), **prottbl**(4)

SEE ALSO

dtime_mgmt(8), **gipc**(8), **demo_client**(8), rthreads and synchronizer papers.

(http://www.opengroup.org/RI/PubProjPgs/CORDS.htm)

NOTES

The ballsorter demo requires CORDS libraries specially compiled to use the *rthreads* real-time threads package. For details, see page 92 of *The CORDS Book*:

ftp://riftp.opengroup.org/pub/real_time_papers/xk_prog_man2.ps

demo_client(8)

NAME

demo_client - The "marker" demonstration

SYNOPSIS

demo_client [**-a** address]

The **demo_client** program is an example application that demonstrates the use of CORDS group communication services.

FLAGS

-a address
: Least significant octet of IP address (in decimal format). (Required when running in single user mode. The more significant portion of the IP addresses of the machines running the demonstration must all be the same.)

DESCRIPTION

The **demo_client** is a non-trivial sample application using CORDS group communication services.

The **demo_client** shows how the system maintains a consistent view of group membership and solves the consensus problem in the presence of various node or link failures.

As each node joins the group, its unique node_id (IP address) determines its position with respect to the other nodes in the group.

Visually this is presented as a ring or track, where the edge nodes (highest and lowest nodes) are the turns and the middle nodes are the straightaways. The ring reconfigures itself in response to membership changes.

A marker moves around the ring, owned by each node in turn, first in increasing (left to right along the bottom) then in decreasing (right to left along the top) order of node_id. The marker steps through nine positions on each node, posting a count-down message via the GIPC service's reliable broadcast mechanism. When the '0' count-down is posted, the next node on the ring owns the marker.

The marker is followed by an anti-marker, which lags it by 180 degrees and moves in step with it. It is drawn in response to a count-down message indicating the location of the marker. The position of the anti-marker, and thus of the node which owns responsibility for drawing (or no longer drawing) the anti-marker can change when a node joining or leaving the group changes the shape of the ring. The anti-marker is guaranteed to appear at the correct position in at most one step of the marker.

If the node which owns the marker leaves the group, the anti-marker becomes the marker and the new anti-marker appears opposite the new marker. If the marker and anti-marker are on the same node, the marker is reassigned to the node closest to its last position and the anti-marker reappears opposite.

Because the posts and membership information are ordered with a global sequence number, the nodes cannot disagree about the order in which nodes join or leave the group, or the location of the marker or how the transition of the marker from one node to the next is ordered with respect to a membership change. A disagreement would be very visible as an inconsistency in the behavior of the marker and/or anti-marker as they move around the ring.

SEE ALSO

gipc(8), **balldemo**(8)

NOTES

This example application has been incorporated into the **balldemo**(8).

dtime_mgmt(8)

NAME

dtime_mgmt - distributed clock management server (for the LTS protocol)

SYNOPSIS

dtime_mgmt

The **dtime_mgmt** server manages LTS.

FLAGS

Configuration parameters are under control of the **rom** file.

DESCRIPTION

The **dtime_mgmt** program is the user-space server which manages distributed clocks synchronized by LTS.

LTS provides a distributed clock with a guaranteed upper bound on the difference between the master clock and any of its client clocks. Client applications access a distributed clock value via the usual MK kernel clock interfaces. LTS is highly configurable, allowing the user to make appropriate tradeoffs among the desired bound, network load and resilience of the service to transient failures.

The following section describes the configuration options.

Configuration parameters are given in the **rom** file. The dtime_mgmt protocol has three rom file options.

<u>dtime_mgmt</u> <u>config</u> [ref_clock_id] ip_address [port]
: Describes the configuration and location of the master clock(s) for this node.

<u>dtime_mgmt</u> <u>provide</u> local_clock_id [port]
: Directs LTS to make the local clock value available to the network as a master clock.

<u>dtime_mgmt</u> <u>local</u> local_clock_id ref_clock_id C1 C2 C3 C4 C5 C6
: Synchronize local clock to reference clock using the parameters C1-C6.

On a given node, a clock may act as both a client clock and a reference clock.

Appropriate values for the rom file options depend quite strongly on the characteristics of the local network and on the desired synchronization properties. The discussion below describes the options in more detail, in general, some experimentation will be needed.

dtime_mgmt config [ref_clock_id] ip_address [port]
: Gives the IP address and the UDP port for the reference clock. The default port is 7123. The default reference clock id is 0 (REALTIME, see mach/clock_types.h for list). The reference clock id specifies both the identity of the clock as an LTS reference and a clock id on the master node.

dtime_mgmt provide local_clock_id [port]
: The specified local clock id is also an LTS reference clock with the same id. LTS makes the clock available on the specified UDP port (default 7123). For a given ref_clock_id/ip_address pair, the rom file on the specified node must indicate that that clock is indeed provided.

dtime_mgmt local local_clock_id ref_clock_id C1 C2 C3 C4 C5 C6
: The specified local clock is synchronized with the specified reference clock with the given parameters. The local clock must be one of the distributed clock objects (DIST_CLOCK_0==3, DIST_CLOCK_1==4). The address of the reference clock is obtained from the matching config line.

The following parameters represent properties of the machines and network and must be set accurately.

C1 = drift_const
: Upper bound on the relative clock drift, r, between the master and client clocks. Expressed as the integer (1 / r)(1 - r). If LTS receives two consecutive values from the master clock which imply that the clocks are drifting at a greater rate, the service will fail. Also note that if this value is incorrectly specified, LTS may silently fail; that is, report as valid a distributed clock value that violates the correctness guarantee.

C2 = net_min
: Minimum one-way request/response time, in microseconds. In practice, half the minimum observed CORDS UDP round trip time is a reasonable choice. If LTS detects a request/response round trip time that is less that twice this value, the service will fail.

The following parameters are used to control the process by which the client reads the master clock.

C3 = net_max
: Upper limit on the one-way request response time. Client clock ignores any response which takes more than twice this long to receive from the master.

C4 = attempts

Number of requests which may made in the course of attempting to reach rapport” with the master clock. If no request results in a timely response, the service must fail.

C5 = attempt_ival

Interval, in milliseconds, at which requests are made.

These parameters establish a limit on the accuracy with which the client clock can read the master clock and the probability with which it will do so.

Allowing for the worst case zero-length interval between clock reads (not network requests!), gives the smallest possible bound on the difference between the master and client clocks.

C6 = bound

The user specifies a bound, in microseconds, on the difference between the reference and client clocks. If LTS cannot ensure this bound, the client clock explicitly fails. The user-specified bound must be greater than or equal to the smallest bound which can be obtained in the configuration specified by C1-C5. The minimum bound will be approximately ((C4 * C5 * 1000) / C1) + (C3 - C2). Note that this means that the configuration is over constrained. If the parameters are inconsistent, an error message is displayed.

FILES

rom(4), **prottbl**(4), **mach/clock_types.h**

gipc(8)

NAME

gipc - group interprocess communication server

SYNOPSIS

The GIPC server provides group interprocess communication services.

FLAGS

Options are provided in the **rom** file or at kernel compile time.

heartbeat rate_usecs rate

The parameter rate is the period, in microseconds, at which the heartbeat protocol sends heartbeat messages.

censustaker tolerance time

The parameter time is the interval, in microseconds, after which, hearing no heartbeat message, the censustaker protocol declares a node to be dead.

In the usual configuration, these parameters should have the same values on every node within the group.

(Note: The GIPC server can also be configured with other protocols in user space. See the CORDS manual).

The GIPC server also accepts the standard framework **rom** options.

FILES

rom(4), **prottbl**(4), sample **rom** file located in src/mach_services/servers/gipc/xkern/gen/rom

SEE ALSO

Travostino, Franco. “Elementary Groupware for Fault Management”. CORDS manual

udpiptest(8)

NAME

udpiptest - send UDP datagram packets between a pair of nodes

SYNOPSIS

udpiptest

DESCRIPTION

The **udpiptest** utility uses the CORDS dgramtest protocol to test the CORDS UDP ping-pong protocol. It is useful both for performance measurement and as a general network diagnostic.

FILES

mach_services/cmds/cords_tests
location of the source code

prottbl protocol table (location can be specified by the ROM file)

ROM file
please refer to the document "The x-kernel-based Communication Subsystem for OSF Mach: changes to the x-kernel 3.2 API", Appendix A for complete details.

A sample ROM file is shown below:

```
#
# Example of ROM file
#

#
# my system and my network
#
arp 123.45.6.78 0:0:C0:36:D0:4C lock
ip gateway 123.45.6.1
#
# about this test
#
dgramtest server 123.45.6.78
dgramtest isserver
#
# or, "dgramtest isclient"
```

#

RELATED INFORMATION

abcasttest(8)

Microkernel Support

latency(1)

NAME

latency - measure interrupt latency and preemption latency

SYNOPSIS

latency [-**p**n:m] [-**t**duration] [-**r**]

FLAGS

-**p**n:m Set the latency test thread's policy and priority. n is a decimal number encoding scheduling policy. The legal values: 1 is Timesharing; 2 is Fixed-priority-RR; 4 is Fixed-priority-FIFO. m is a decimal number encoding priority.

-**t**duration
The duration for running latency. 1s <= duration <= 9999s. Default is 60 seconds.

-**r** If this switch is specified, events occur on random system clock ticks. (By default, a system clock tick occurs every 10 milliseconds.) Otherwise, events occur uniformly once every 10 clock ticks.

DESCRIPTION

The latency test is one of the tests that measures operating system performance by measuring response time to external events. This is a traditional benchmark for real-time systems.

Typical latency test conditions: run AIMIII in background, wait a specified time (usually one minute) to fill up the cache and start the processes. By default, the system clock frequency is 100. Each event is created every 100ms.

EXAMPLES

Run **latency** without AIMIII load during 2 minutes (1200 events) in Fixed Round Robin at priority 10. The result is saved into a data file named ‘lat.dat’.

latency -**p**2:10 -**t**120 > lat.dat

sysversion(1)

NAME

sysversion - print microkernel and OSF/1 server version strings and server collocation status.

SYNOPSIS

sysversion

DESCRIPTION

The **sysversion** command writes to the standard output the version strings for the microkernel and OSF/1 server. Using the **table**(2) server call, sysversion also reads the OSF/1 server variable that indicates whether or not the server is running collocated with the microkernel. If the value of this variable is nonzero, **sysversion** prints the message “server is kernel loaded.” If this value is zero, it prints the message “server is not kernel loaded” instead.

EXIT VALUE

If any of the calls to **table**(2) succeed, **sysversion** returns with a 0 exit value. Otherwise it returns with an exit value of 1.

RELATED INFORMATION

system calls: **table**(2)

IP Multicast

IP Multicasting[1]

IP multicasting is currently supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW, and only on subnetworks for which the interface driver has been modified to support multicasting.

The next subsections describe how to send and receive multicast datagrams.

Sending IP Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in a **sendto (2)** call.

The definitions required for the multicast-related socket options are found in <netinet/in.h>. All IP addresses are passed in network byte-order.

By default, IP multicast datagrams are sent with a time-to-live (TTL) of 1, which prevents them from being forwarded beyond a single subnetwork. A new socket option allows the TTL for subsequent multicast data-

---

1. Copyright © 8.2 (Berkeley) 3/21/94

grams to be set to any value from 0 to 255, in order to control the scope of the multicasts:

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Multicast datagrams with a TTL of 0 will not be transmitted on any subnet, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket (see below). Multicast datagrams with TTL greater than one may be delivered to more than one subnet if there are one or more multicast routers attached to the first-hop subnet. To provide meaningful scope control, the multicast routers support the notion of TTL "thresholds", which prevent datagrams with less than a certain TTL from traversing certain subnets. The thresholds enforce the following convention:

Scope	**Initial TTL**
restricted to the same host	1
restricted to the same subnet	32
restricted to the same site	64
restricted to the same continent	128
unrestricted	255

"Sites" and "regions" are not strictly defined, and sites may be further subdivided into smaller administrative units, as a local matter.

An application may choose an initial TTL other than the ones listed above. For example, an application might perform an "expanding-ring search" for a network resource by sending a multicast query, first with a TTL of 0, and then with larger and larger TTLs, until a reply is received, perhaps using the TTL sequence 0, 1, 2, 4, 8, 16, 32.

The multicast router **mrouted (8)**, refuses to forward any multicast datagram with a destination address between 224.0.0.0 and 224.0.0.255, inclusive, regardless of its TTL. This range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting.

The address 224.0.0.0 is guaranteed not to be assigned to any group, and 224.0.0.1 is assigned to the permanent group of all IP hosts (including gateways). This is used to address all multicast hosts on the directly connected network. There is no multicast address (or any other IP address) for all hosts on the total Internet. The addresses of other well-known, permanent groups are published in the “Assigned Numbers” RFC, which is available from the InterNIC.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. (If the host is also serving as a multicast router, a multicast may be *forwarded* to interfaces other than originating interface, provided that the TTL is greater than 1.) The default interface to be used for multicasting is the primary network interface on the system. A socket option is available to override the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

where ***addr*** is the local IP address of the desired outgoing interface. An address of INADDR_ANY may be used to revert to the default interface. The local IP address of an interface can be obtained via the SIOCGIFCONF ioctl. To determine if an interface supports multicasting, fetch the interface flags via the SIOCGIFFLAGS ioctl and see if the IFF_MULTICAST flag is set. (Normal applications should not need to use this option; it is intended primarily for multicast routers and other system services specifically concerned with internet topology.) The SIOCGIFCONF and SIOCGIFFLAGS ioctls are described in the previous section.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

where ***loop*** is set to 0 to disable loopback, and set to 1 to enable loopback. This option improves performance for applications that may have no more than one instance on a single host (such as a router demon), by eliminating the overhead of receiving their own transmissions. It should generally not be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

Receiving IP Multicast Datagrams

Before a host can receive IP multicast datagrams, it must become a member of one or more IP multicast groups. A process can ask the host to join a multicast group by using the following socket option:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq,
           sizeof(mreq))
```

where ***mreq*** is the following structure:

```
struct ip_mreq {
        struct in_addr imr_multiaddr; /* multicast group to join */
        struct in_addr imr_interface; /* interface to join on */
}
```

Every membership is associated with a single interface, and it is possible to join the same group on more than one interface. "imr_interface" should be INADDR_ANY to choose the default multicast interface, or one of the host's local addresses to choose a particular (multicast-capable) interface. Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.

To drop a membership, use:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq,
           sizeof(mreq));
```

where ***mreq*** contains the same values as used to add the membership. The memberships associated with a socket are also dropped when the socket is closed or the process holding the socket is killed. However, more than one socket may claim a membership in a particular group, and the host will remain a member of that group until the last claim is dropped.

The memberships associated with a socket do not necessarily determine which datagrams are received on that socket. Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram; however, delivery of a multicast datagram to a particular socket is based on the destination port (or protocol type, for raw sockets), just as with unicast datagrams.

To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified (i.e., INADDR_ANY). To receive multicast datagrams sent to a particular group and port, bind to the local port, with the local address set to the multicast group address. Once bound to a multicast address, the socket cannot be used for sending data.

More than one process may bind to the same SOCK_DGRAM UDP port or the same multicast group and port if the **bind** call is preceded by:

```
int on = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on));
```

All processes sharing the port must enable this option. Every incoming multicast or broadcast UDP datagram destined to the shared port is delivered to all sockets bound to the port. For backwards compatibility reasons, this does not apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port.

A final multicast-related extension is independent of IP: two new ioctls, SIOCADDMULTI and SIOCDELMULTI, are available to add or delete link-level (e.g., Ethernet) multicast addresses accepted by a particular interface. The address to be added or deleted is passed as a sockaddr structure of family AF_UNSPEC, within the standard ifreq structure.

These ioctls are for the use of protocols other than IP, and require superuser privileges. A link-level multicast address added via SIOCADDMULTI is not automatically deleted when the socket used to add it goes away; it must be explicitly deleted. It is inadvisable to delete a link-level address that may be in use by IP.

Sample Multicast Program

The following program sends or receives multicast packets. If invoked with one argument, it sends a packet containing the current time to an arbitrarily-chosen multicast group and UDP port. If invoked with no arguments, it receives and prints these packets. Start it as a sender on just one host and as a receiver on all the other hosts.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <stdio.h>

#define EXAMPLE_PORT 60123
#define EXAMPLE_GROUP "224.0.0.250"

main(argc)
 int argc;
{
        struct sockaddr_in addr;
        int addrlen, fd, cnt;
```

```
struct ip_mreq mreq;
char message[50];

fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd < 0) {
        perror(“socket”);
        exit(1);
}

bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(EXAMPLE_PORT);
addrlen = sizeof(addr);

if (argc > 1) { /* Send */
        addr.sin_addr.s_addr = inet_addr(EXAMPLE_GROUP);
        while (1) {
                time_t t = time(0);
                sprintf(message, “time is %-24.24s”, ctime(&t));
                cnt = sendto(fd, message, sizeof(message), 0,
                        (struct sockaddr *)&addr, addrlen);
                if (cnt < 0) {
                        perror(“sendto”);
                        exit(1);
                }
                sleep(5);
        }
} else { /* Receive */
        if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
                perror(“bind”);
                exit(1);
        }

        mreq.imr_multiaddr.s_addr =
                    inet_addr(EXAMPLE_GROUP);
        mreq.imr_interface.s_addr = htonl(INADDR_ANY);
        if (setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
        &mreq, sizeof(mreq)) < 0) {
                perror(“setsockopt mreq”);
                exit(1);
        }

        while (1) {
                cnt = recvfrom(fd, message, sizeof(message), 0,
                        (struct sockaddr *)&addr, &addrlen);
               if (cnt <= 0) {
                       if (cnt == 0) {
                               break;
                       }
                        perror(“recvfrom”);
                        exit(1);
```

```
                }
                printf(“%s: message = …\”%s…\”…\n”,
                inet_ntoa(addr.sin_addr), message);
            }
        }
}
```