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

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

DESCRIPTION

The **mach_port_status** structure 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

```
struct routine_arg_descriptor
{
        routine_arg_type        type;
        routine_arg_size        size;
        routine_arg_size        count;
        routine_arg_offset      offset;
} *routine_arg_descriptor_t;
```

DESCRIPTION

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

FIELDS

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

size
The element size, in bytes.

count
The number of elements.

offset
Offset in the list of routine arguments.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_descriptor, rpc_subsystem**.

routine_descriptor

Structure — Defines the nature of each subsystem routine

SYNOPSIS

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

DESCRIPTION

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

FIELDS

impl_routine
Server function pointer (code entry point).

stub_routine
Unmarshalling function pointer.

argc
Number of argument words.

descr_count
Number of complex argument descriptors.

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

max_reply_msg
The maximum size for the reply message.

RELATED INFORMATION

Functions: **mach_subsystem_create.**

Structures: **routine_arg_descriptor, rpc_subsystem**.

rpc_subsystem

Structure — Defines the set of server routines in a subsystem

SYNOPSIS

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


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

DESCRIPTION

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

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

FIELDS

subsystem
: Reserved for system use.

start
: Minimum routine number.

end
: Maximum routine number (+1).

maxsize

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

base_addr

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

routine

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

arg_descriptor

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

RELATED INFORMATION

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

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

Device Driver

device_get_status

Function — Return the current device status

LIBRARY

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

SYNOPSIS

kern_return_t **device_get_status**

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

DESCRIPTION

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

PARAMETERS

device

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

flavor

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

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

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

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

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

status

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

status_count

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

RETURN VALUE

D_DEVICE_DOWN

Device has been shut down

D_INVALID_OPERATION

The device driver does not export any semaphores.

D_NO_SUCH_DEVICE

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

D_OUT_OF_BAND

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

RELATED INFORMATION

Functions: **device_set_status**.

Structures: *flavor* dependent.

device_read_async

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

SYNOPSIS

kern_return_t **device_read_async**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_read_async returns only invalid parameter errors.

RELATED INFORMATION

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

device_read_async_inband

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

SYNOPSIS

kern_return_t **device_read_async_inband**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_read_async_inband returns only invalid parameter errors.

RELATED INFORMATION

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

device_read_overwrite_async

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

SYNOPSIS

kern_return_t **device_read_overwrite_async**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

bytes_wanted
: [in scalar] Size of data transfer.

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

RETURN VALUE

device_read_overwrite_async returns only invalid parameter errors.

RELATED INFORMATION

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

device_write_async

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

SYNOPSIS

kern_return_t **device_write_async**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

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

bytes_wanted
: [in scalar] Size of data transfer.

RETURN VALUE

device_write_async returns only invalid parameter errors.

RELATED INFORMATION

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

device_write_async_inband

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

SYNOPSIS

kern_return_t **device_write_async_inband**

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

DESCRIPTION

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

PARAMETERS

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

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

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

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

  D_NOWAIT
  : Do not wait if data is unavailable.

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

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

bytes_wanted
[in scalar] Size of data transfer.

RETURN VALUE

device_write_async_inband returns only invalid parameter errors.

RELATED INFORMATION

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

io_done_queue_create

Function — Create an io_done_queue kernel object

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

KERN_INVALID_ARGUMENT
The *host* argument is not valid.

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

KERN_SUCCESS
The io_done_queue was created.

RELATED INFORMATION

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

io_done_queue_terminate

Function — Terminate an io_done_queue kernel object

SYNOPSIS

```
kern_return_t io_done_queue_terminate
        (mach_port_t                    queue);
```

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

Only generic errors apply.

RELATED INFORMATION

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

io_done_queue_wait

System Trap— Wait on an io_done_queue kernel object

SYNOPSIS

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

DESCRIPTION

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

PARAMETERS

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

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

RETURN VALUE

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

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

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

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

RELATED INFORMATION

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

Data Structures: **io_done_result**.

io_done_result

Structure — Defines asynchronous I/O completion status

SYNOPSIS

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

DESCRIPTION

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

FIELDS

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

IO_DONE_READ The operation was a device_read_async or device_read_async_inband.

IO_DONE_OVERWRITE The operation was a device_read_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.

EINVALThe *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.

dtime_mgmt config [ref_clock_id] ip_address [port]
: Describes the configuration and location of the master clock(s) for this node.

dtime_mgmt provide local_clock_id [port]
: Directs LTS to make the local clock value available to the network as a master clock.

dtime_mgmt local 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);
                }
        }
}
```