Message Header

The C type definition for the message header is as follows (from the header file **mach/message.h**):

```
typedef struct {
        unsigned int msg_unused : 24,
                     msg_simple : 8;
        unsigned int msg_size;
        int          msg_type;
        port_t       msg_local_port;
        port_t       msg_remote_port;
        int          msg_id;
msg_header_t;¬} msg_header_t;
```

The **msg_simple** field indicates whether the message

is *simple* or *nonsimple*; the message is simple if its body contains neither ports nor out-of-line data (pointers).

The **msg_size** field specifies the size of the message to be sent, or the maximum size of the message that can be received. When a message is received, Mach sets **msg_size** to the size of the received message. The size includes the header and in-line data and is given in bytes.

The **msg_type** field specifies the general type of the message. For hand-built messages, it's MSG_TYPE_NORMAL; MiG-generated servers use the type MSG_TYPE_RPC. Other values for the

msg_type field are defined in the header files **mach/message.h** and **mach/msg_type.h**.

The **msg_local_port** and **msg_remote_port** fields name the ports on which a message is to be received or sent. Before a message is sent, **msg_local_port** must be set to the port to which a reply, if any, should be sent; **msg_remote_port** must specify the port to which the message is being sent. Before a message is received, **msg_local_port** must be set to the port or port set to receive on. When a message is received, Mach sets **msg_local_port** to the port the message is received on, and **msg_remote_port** to the port any reply should be sent to (the sender's **msg_local_port**).

The **msg_id** field can be used to identify the meaning of the message to the intended recipient. For example, a program that can send two kinds of messages should set the **msg_id** field to indicate to the receiver which kind of message is being sent. MiG automatically generates values for the **msg_id** field.

Message Body

The body of a message consists of an array of type descriptors and data. Each type descriptor contains

the following structure:

```
typedef struct  {
     // Type of data
     unsigned int msg_type_name : MSG_TYPE_BYTE,
     // Number of bits per item
     msg_type_size : 8,
     // Number of items
     msg_type_number : 12,
     // If true, data follows; else a ptr to data
follows
     msg_type_inline : 1,
     // Name, size, number follow
     msg_type_longform : 1,
     // Deallocate port rights or memory
     msg_type_deallocate : 1,
     msg_type_unused : 1;
```

```
msg_type_t;¬} msg_type_t;
```

The **msg_type_name** field describes the basic type of data comprising this object. The system-defined data types include:

· Ports, including combinations of send and receive rights.

· Port and port set names. This is the same language data type as port rights, but the message only carries a task's name for a port and doesn't cause any transferral of rights.

· Simple data types, such as integers, characters, and floating-point values.

The **msg_type_size** field indicates the size in bits of the basic object named in the **msg_type_name** field.

The **msg_type_number** field indicates the number of items of the basic data type present after the type descriptor.

The **msg_type_inline** field indicates that the actual data is included after the type descriptor; otherwise, the word following the descriptor is a pointer to the data to be sent.

The **msg_type_longform** field indicates that the name, size, and number fields were too long to fit into the **msg_type_t** structure. These fields instead follow the **msg_type_t** structure, and the type

descriptor consists of a **msg_type_long_t**:

```
typedef struct {
    msg_type_t  msg_type_header;
    short       msg_type_long_name;
    short       msg_type_long_size;
    int         msg_type_long_number;
} msg_type_long_t;
```

When **msg_type_deallocate** is nonzero, it indicates that Mach should deallocate this data item from the sender's address space after the message is queued. You can deallocate only port rights or out-of-line data.

A data item, an array of data items, or a pointer to data follows each type descriptor.

Setting Up a Simple Message

As described earlier, a message is *simple* if its body doesn't contain any ports or out-of-line data (pointers). The **msg_remote_port** field must contain the port the message is to be sent to. The **msg_local_port** field should be set to the port a reply message (if any) is expected on.

The following example shows the creation of a simple message. Because every item in the body of the message is of the same type (**int**), only one type descriptor is necessary, even though the items are in

two different fields.

```
#define BEGIN_MSG 0 /* Constants to identify the


     different messages */
#define END_MSG 1
#define REPLY_MSG 2

#define MAXDATA 3

struct simp_msg_struct {
     msg_header_t   h;   /* message header */
     msg_type_t     t;              /* type descriptor
*/
     int  inline_data1;    /* start of data array*/
     int  inline_data2[2];
```

```
};
struct simp_msg_struct  msg_xmt;
port_t                  comm_port, reply_port;

/* Fill in the message header. */
msg_xmt.h.msg_simple = TRUE;
msg_xmt.h.msg_size = sizeof(struct simp_msg_struct);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_local_port = reply_port;
msg_xmt.h.msg_remote_port = comm_port;
msg_xmt.h.msg_id = BEGIN_MSG;

/* Fill in the type descriptor. */
msg_xmt.t.msg_type_name = MSG_TYPE_INTEGER_32;
msg_xmt.t.msg_type_size = 32;
msg_xmt.t.msg_type_number = MAXDATA;
msg_xmt.t.msg_type_inline = TRUE;
```

```
msg_xmt.t.msg_type_longform = FALSE;
msg_xmt.t.msg_type_deallocate = FALSE;

/* Fill in the array of data items. */
msg_xmt.inline_data1 = value1;
msg_xmt.inline_data2[1] = value2;
msg_xmt.inline_data2[2] = value3;
```

port_allocate()

SUMMARY Create a port

SYNOPSIS **#import <mach/mach.h>**

port_allocate;¬kern_return_t **port_allocate(**task_t

task, port_name_t ***port_name*)**

ARGUMENTS *task*: The task in which the new port is created (for example, use **task_self()** to specify the caller's task).

port_name: Returns the name used by *task* for the new port.

DESCRIPTION The function **port_allocate()** causes a port to be created for the specified task; the resulting port is returned in *port_name.* The target task initially has both send and receive rights to the port. The new port isn't a member of any port set.

EXAMPLE

```
port_t          myport;
kern_return_t   error;

if ((error=port_allocate(task_self(), &myport)) !=
KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}
```

RETURN KERN_SUCCESS: A port has been allocated.

KERN_INVALID_ARGUMENT: *task* was invalid.

KERN_RESOURCE_SHORTAGE: No more port slots are available for this task.

DPSAddPort;¬void **DPSAddPort(**port_t *port*, DPSPortProc *handler*, int *maxMsgSize*, void **userData*, int *priority***)**
void **DPSRemovePort(**port_t *port***)**

DESCRIPTION **DPSAddPort()** registers the function *handler* to be called each time your application asks for an event or peeks at the event queue. The function is called provided the following are true:

· The Mach port *port* must be valid and it must hold a message waiting to be read.

· *priority*, an integer from 0 to 30, must be equal

to or greater than the application's current priority threshold. See **DPSAddTimedEntry()** for a further explanation.

DPSPortProc, *handler*'s defined type, takes the form

```
void *handler(msg_header_t *msg, void *userData)
```

where *msg* is a pointer to the message that was received at the port and *userData* is the same pointer that was passed as the fourth argument to **DPSAddPort()**. The *userData* pointer is provided as a convenience, allowing you to pass arbitrary data to *handler*.

#import <mach/cthreads.h>

cthread_fork;¬cthread_t **cthread_fork(**any_t (**function*)(), any_t *arg***)**

DESCRIPTION The function **cthread_fork()** takes two arguments: a function for the new thread to execute, and an argument to this function. The **cthread_fork()** function creates a new thread of control in which the specified function is executed concurrently with the caller's thread. This is the sole means of creating new threads.

The **any_t** type represents a pointer to any C type. The **cthread_t** type is an integer-size handle that uniquely identifies a thread of control. Values of type

cthread_t will be referred to as thread identifiers. Arguments larger than a pointer must be passed by reference. Similarly, multiple arguments must be simulated by passing a pointer to a structure containing several components. The call to **cthread_fork()** returns a thread identifier that can be passed to **cthread_join()** or **cthread_detach()**. Every thread must be either joined or detached exactly once.

cthread_abort;¬kern_return_t
cthread_abort(cthread_t *t***)**

DESCRIPTION This function provides the functionality of

thread_abort() to C threads. The **cthread_abort()** function interrupts system calls; it's usually used along with **thread_suspend()**, which stops a thread from executing any more user code. Calling **cthread_abort()** on a thread that isn't suspended is risky, since it's difficult to know exactly what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

See **thread_abort()** for a full description of the use of this function.

thread_set_state;¬kern_return_t

thread_set_state(thread_t *target_thread*, int *flavor*, thread_state_data_t *new_state*, unsigned int *new_state_count***)**

DESCRIPTION The function **thread_get_state()** returns the state component (that is, the machine registers) of *target_thread* as specified by *flavor.* The *old_state* is an array of integers that's provided by the caller and returned filled with the specified information. You should set *old_state_count* to the maximum number of integers in *old_state*. On return, *old_state_count* is equal to the actual number of integers in *old_state*.

The function **thread_set_state()** sets the state

component of *target_thread* as specified by *flavor.* The *new_state* is an array of integers that the caller fills. You should set *new_state_count* to the number of elements in *new_state.* The entire set of registers is reset.

target_thread must not be **thread_self()** for either of these calls.

The state structures are defined in the header file **mach/machine/thread_status.h**.

AppKitProgramming;¬AppKit Programming

AppKit_Application;¬Class: Application workspace;¬workspace

+ (id <NXWorkspaceRequestProtocol>)**workspace**

Returns the Workspace Manager. You need that in order to send it a message asking it to do such things as open a file. The Workspace Manager responds to the **NXWorkspaceRequest** protocol. Here's an example of asking the Workspace Manager for the icon for the file "x.draw":

```
NXImage *i = [[Application workspace]
getIconForFile:"x.draw"];
```