2

Using Mach Messages

This chapter describes how to use Mach messages for interprocess communication (IPC). Programs can either send and receive Mach messages directly, or they can use remote procedure calls (RPCs) generated by MiG (Mach Interface Generator). MiG-generated RPCs appear to be simple function calls but actually involve messages. Many kernel functions, such as **host_info()**, are really RPCs.

This chapter first describes the structure of all messages. It then discusses how to set up messages for direct sending. Finally, it discusses how to use MiG to build a *Mach server*Ða program that provides services to clients by using remote procedure calls. This chapter assumes that you understand the concepts of ports, port sets, and messages, which are described in Chapter€1, ªMach Concepts.º

You should usually use MiG to generate messages. MiG-generated code is easier for clients to use, and using MiG is a good way to define an interface that's separate from the implementation. However, you might want to build messages by hand if the messages are very simple or if you want fine control over communication details.

Note: Tasks can also communicate with each other using Distributed Objects. See the *NeXTSTEP General Reference* for information on Distributed Objects.

Message Structure

A message consists of a fixed header often followed by the message body. The body consists of alternating type descriptors and data items. Here's a typical message structure:

```
typedef struct {
    msg_header_t Head;
    msg_type_t   aType;
    int          a;
    msg_type_t   bType;
    int          b;
} Request;
```

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;
```

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

Creating Messages by Hand

This section shows how to create messages to be sent using **msg_send()** or **msg_rpc()**. You don't usually have to set up messages by hand. For example, although Mach servers call **msg_send()**, almost all the message fields are already set up in MiG-generated code. However, this section might be useful if you want to send messages without using MiG, or if you want to read through MiG-generated code.

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;
```

Setting Up a Nonsimple Message

A message is *nonsimple* if its body contains ports or out-of-line data. The most common reason for sending data out-of-line is that the data block is very large or of variable size.

In-line data is copied by the sender into the message structure and then often copied out of the message by the receiver. *Out-of-line data*, however, is mapped by the kernel from the address space of the sender to the address space of the receiver. No actual copying of out-of-line data is done unless one of the two tasks subsequently modifies the data.

This example shows how to construct a message containing out-of-line data:

```
#define BEGIN_MSG 0 /* Constants to identify the different messages */
#define END_MSG 1
#define REPLY_MSG 2

#define MAXDATA 3

struct ool_msg_struct {
    msg_header_t  h;                /* message header */
    msg_type_t    t;                /* type descriptor */
    int          *out_of_line_data; /* address of data */
};
struct ool_msg_struct   msg_xmt;
port t                  comm port, reply port;

/* Fill in the message header. */
msg_xmt.h.msg_simple = FALSE;
msg_xmt.h.msg_size = sizeof(struct ool_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 = FALSE;
msg_xmt.t.msg_type_longform = FALSE;
msg_xmt.t.msg_type_deallocate = FALSE;

/* Fill in the out-of-line data. */
msg_xmt.out_of_line_data = (int *)&mydata;
```

The fields that change values from those in the simple message example are **msg_simple**, **msg_type_inline**, and possibly **msg_type_deallocate**. The **msg_type_name**, **msg_type_size**, and **msg_type_number** fields remain the same as before, so that Mach can determine how much memory to map.

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 where a reply message (if any) is expected.

Setting Up a Reply Message

Once a message has been received, a reply message may have to be sent to the sender of the received message. In

the following example, the reply message, **msg_xmt**, is simply a **msg_header_t** since no data is required. The **msg_remote_port** field, which designates where to send the message, must be set to the remote port of the previously received message (which Mach set to the previous sender's **msg_local_port** field). The **msg_local_port** field of the outgoing message is set to PORT_NULL because no reply to this message is expected.

```
#define BEGIN_MSG 0 /* Constants to identify the different messages */
#define END_MSG 1
#define REPLY_MSG 2

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

msg_xmt.h.msg_remote_port = msg_rcv->h.msg_remote_port;
msg_xmt.h.msg_local_port = PORT_NULL;  /* no reply expected */
msg_xmt.h.msg_id = REPLY_MSG;
msg_xmt.h.msg_size = sizeof(msg_header_t);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_simple = TRUE;
```

Mach Interface Generator

The Mach Interface Generator (known as MiG) is a program that generates remote procedure call (RPC) code for communication between a client and a server process. The operations of sending a message and receiving a reply are represented as a single remote procedure call.

For example, if a program makes a call to **host_info()**, it actually calls a library routine that sends a message to the Mach kernel and then waits to receive a reply message. After the Mach kernel sends a reply message containing the information, the library routine takes the data out of the reply message and returns it to the program in parameters to the **host_info()** call. However, the program sees none of this complexityÐit merely makes the following function call:

```
ret = host_info(host_self(), HOST_SCHED_INFO,
                (host_info_t)&sched_info, &sched_count);
```

A Mach server executes as a separate task and communicates with its clients by sending Mach messages. As you can see from the previous sections in this chapter, Mach messages are fairly complex. The MiG program is designed to automatically generate procedures in C to pack and send, or receive and unpack the messages used to communicate between processes.

Because of the complexity of sending and decoding messages, Mach remote procedure calls are an order of magnitude slower than real function calls, even if the server is on the local machine. Calls to servers on remote machines take longer. However, Mach RPC has the advantages of the separation of interface and implementation, and of network transparency.

Using MiG, you can create RPC interfaces for sending messages between tasks on the local machine, or between tasks on separate machines in a network. In the network environment, MiG both encodes messages to be transmitted and decodes them upon arrival at the destination node, taking into account dissimilarities in machine architecture.

MiG is especially useful if you're faced with a mixed network environment. Without MiG, you're responsible for providing routines to translate messages between two machines with different data representations. Using MiG, you need only specify the calling arguments of the procedure and the procedure's return variables. The low-level routines required to translate messages between these machines are then generated automatically.

MiG is flexible enough to describe most data structures that might be sent as messages between processes. MiG supports the data types boolean, character, signed and unsigned integers, integer subranges, strings, reals, and communication port types. MiG also supports the limited creation of new data types through the use of enumerations, fixed-size and variable-size arrays, records, pointers to these types, and unions.

Creating Mach Servers with MiG

To create a Mach server, you must provide a specification file defining parameters of both the message-passing interface and the procedure-call interface. MiG then generates three files from the specification file:

- User interface file (*xxx***User.c**, where *xxx* is the subsystem name)ÐShould be compiled and linked into the client program. It implements and exports the procedures and functions for sending and receiving the appropriate messages to and from the server.
- User header file (*xxx***.h**)ÐDefines the functions to be called by a client of the server. It's included in the user interface file (*xxx***User.c**) and defines the types and routines needed at compilation time.
- Server interface file (*xxx***Server.c**)ÐShould be compiled and linked into the server process. It extracts the input parameters from an IPC message, and calls a server procedure to perform the operation. When the server procedure or function returns, the Server interface also gathers the output parameters and formats a reply message.

Besides the specification file, you must write at least two functions for the Mach server. One is the main routine of the server, which registers the server and then goes into a loop that receives a message, calls the MiG-generated code to process the request, and sends a reply message. You must also write one function for each remote procedure call, so that the MiG-generated server code can call the appropriate function for each request.

In addition, you should provide a library routine that clients can use to look up your server. For example, the kernel-server loader has a routine called **kern_loader_look_up()** that clients call to obtain the kernel-server loader's port. This port must be specified as the first argument in every RPC to the kernel-server loader.

You can register your server with either the Network Name Server or the Bootstrap Server, depending on whether you want your server to be available to other machines on a network. The Bootstrap Server allows only processes that are on the local machine (or a subset of local processes) to get your server's port. For example, the sound driver registers its port with the Bootstrap Server so that only processes descended from the local machine's Login Window can control sound. The Network Name Server allows tasks on remote machines to get the server's port. See Chapter€4, ªMach Functions,º for more information on Network Name Server and Bootstrap Server functions.

Client's View

This section describes how clients use servers, so that you can better create and document your own server.

Before a client can make remote procedure calls to the server, it must find the server's port. If the server doesn't provide a library function to do this lookup, then the client must call either **netname_look_up()** or **bootstrap_look_up()** and supply the name of the server.

When a client makes a remote procedure call, it appears to be a simple function call. The return type depends on whether the RPC is defined in the server's MiG specification file to be a routine, procedure, or function (as described later in this chapter).

The most convenient interfaces are to routines, which return a value of type **kern_return_t**. The returned value is either KERN_SUCCESS or a MiG, Mach, or server-specific error code. MiG and Mach error codes can be interpreted by **mach_error()** and **mach_error_string()**.

Procedure and function RPCs are less convenient than routines because they don't directly return error codes. Instead, the client must provide an error-handling routine named either **MsgError()** or whatever name the server

developer specified in the server's MiG specification file. The error-handling routine must be defined as follows:

void *error_proc*(kern_return_t *error_code*)

Common Error Codes

The most common system error that an RPC returns to a client is an invalid port. This can mean several things:

- The request port (usually the first parameter in the RPC) is an invalid port, or the client doesn't have send rights for it.
- The reply port is invalid or lacks receive rights. (This problem can't occur unless the client provides the reply port; usually the system provides it.)
- Another port that the client is passing in the message is invalid.
- A port that's being passed back to the client is invalid.

Another system error a client might receive is a timeout. This can happen only if a timeout is specified in an argument or in the server's specification file, and usually doesn't happen unless the server is on a different machine from the client.

MiG errors, which are defined in the header file **mach/mig_errors.h**, usually occur only if the client is using a different version of the interface than the server.

Out-of-Line Data

When making specific interface calls, the client should be aware if any out-of-line data is being returned to it. If so, it might want to deallocate the space with a call to **vm_deallocate()**.

Compiling the Client

The client must be compiled and linked with the *xxx***User.c** and *xxx***.h** files that MiG produced from the server's specification file. The client should also include or be linked with any files that are necessary to communicate with the server (such as the file containing the routine that looks up the server). For example, clients of the kernel-server loader must€be linked against the kernload library, which supplies all non-RPC kernel-server loader functions.

Programming Example

This example shows the implementation of a simple server that adds two or three integers and returns the answer. The files used to produce this server and a sample client program are under the **MiG** directory of **/NextLibrary/Documentation/NextDev/Examples**.

The user-written files required for the server are the following:

- MiG specification file (**Server/add.defs**)
- Type definition file, which is included by both the server and the client (**Library/add_types.h**)
- Implementation file, which contains the server's main loop and the function that does the addition (**Server/add_server_main.c**)

Once the server has been generated, any client programs need to have the following files:

- MiG-generated user interface file, in a form that can be compiled or linked into the client program (**Library/addUser.o**)

- MiG-generated user header file (**Library/add.h**)
- Type definition file (**Library/add_types.h**)
- One or more files containing the main parts of the client program (**Client/add.c**)

In the following example, a simple MiG specification file called **add.defs** is shown. It declares the remote routines **add2nums()** and **add3nums()**, which take as arguments the request port (the default first argument to every MiG operation), two or three integers, and a pointer to another integer. Because all types mentioned in **add.defs** are already defined in the included header file **mach/std_types.defs**, it isn't necessary to define any types directly in **add.defs**.

```
/* add.defs:  MiG definition file for add server */

subsystem add 0;

/* Get standard definitions of int and port_t. */
#include <mach/std_types.defs>

routine add2nums(server: port_t; a:int; b:int; out c:int);
routine add3nums(server: port_t; a:int; b:int; c:int; out d:int);
```

The header file **mach/std_types.defs** defines **int** and **port_t** as the following:

```
type int = MSG_TYPE_INTEGER_32;
type port_t = MSG_TYPE_PORT;
```

The header file **add_types.h** contains definitions needed by both the client and the server:

```
/* add_types.h:  Definitions for add server */
#import <mach/mach.h>

#define ADD_SERVER_NAME "Addition-Server"

extern port_t add_look_up(void);
```

The code that does the work for the server is in the file **add_server_main.c**. It contains a main loop and the functions that perform the addition. The main loop dynamically allocates the memory needed for incoming and outgoing messages, using the **addMaxRequestSize** and **addMaxReplySize** constants generated by MiG.

```
/* add_server_main.c:  Main loop and implementation of add server */

#import <mach/mach.h>
#import <mach/message.h>
#import <servers/netname.h>
#import <mach/mach_error.h>
#import <ansi/stdlib.h>
#import "../Library/add_types.h"
#import "addServer.h"

void         server_loop(port_t port);
/* defined by MiG: */
boolean_t    add_server(msg_header_t *in, msg_header_t *out);

void main(int argc, char *argv[])
{
    port_t        server_port;
    kern_return_t r;

    /* Register with the Network Name Server. */
    r = port_allocate(task_self(), &server_port);
    if (r != KERN_SUCCESS) {
        mach_error("port_allocate failed", r);
        exit(1);
    }
    r = netname_check_in(name_server_port, ADD_SERVER_NAME,
        PORT_NULL, server_port);
```

```
    if (r != KERN_SUCCESS) {
        mach_error("netname_check_in failed", r);
        exit(1);
    }

    /* Enter our main loop. */
    server_loop(server_port);
}

void server_loop(port_t port)
{
    kern_return_t ret;
    msg_header_t *msg = (msg_header_t *)malloc(addMaxRequestSize);
    msg_header_t *reply = (msg_header_t *)malloc(addMaxReplySize);

    while (TRUE)
    {
        /* Receive a request from a client. */
        msg->msg_local_port = port;
        msg->msg_size = addMaxRequestSize;
        ret = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (ret != RCV_SUCCESS) /* ignore errors */;

        /* Feed the request into the server. */
        (void)add_server(msg, reply);

        /* Send a reply to the client. */
        reply->msg_local_port = port;
        ret = msg_send(reply, MSG_OPTION_NONE, 0);
        if (ret != SEND_SUCCESS) /* ignore errors */;
    }
}

/*
 * This function is called by add_server, which was created by MiG.
 * It is NOT directly called by any client process.
 */
kern_return_t add2nums(port_t server, int n1, int n2, int *n3)
{
    *n3 = n1+n2;
    return KERN_SUCCESS;
}

/*
 * This function is called by add_server, which was created by MiG.
 * It is NOT directly called by any client process.
 */
kern_return_t add3nums(port_t server, int n1, int n2, int n3, int *n4)
{
    *n4 = n1+n2+n3;
    return KERN_SUCCESS;
}
```

In general, your message receive loop should return a reply for every message it receives unless the reply message returned from the MiG-generated server has MIG_NO_REPLY in its **RetCode** field. MIG_NO_REPLY is used only when the received message was part of an RPC that never expects a return message (a **simpleprocedure** or **simplefunction**, both of which are defined later in this chapter). For example:

```
(void)add_server(msg, reply);
ret_code = reply.RetCode;

if (ret_code == MIG_NO_REPLY)
    ret_code = KERN_SUCCESS;
else
    ret_code = msg_send(reply, MSG_OPTION_NONE, 0);
```

Finally, a typical client process, such as **Client/add.c**, makes the RPC as follows:

```
. . .
#import "../Library/add_types.h"
#import "../Library/add.h"
int           n1, n2, n3, result;
kern_return_t ret;
port_t        server;
. . .
/* Find the server. */
server = add_look_up();
if (server == PORT_NULL)
{
    fprintf(stderr, "Couldn't find the add server.\n");
    exit(2);
}

/* Send a message to the server. */
if (argc == 3) {  /* 2 numbers to add */
    ret = add2nums(server, n1, n2, &result);
    if (ret != KERN_SUCCESS)
        printf("Call to add2nums failed.\n");
    else
        printf("According to the server, %d + %d = %d.\n", n1, n2,
            result);
} else {          /* 3 numbers to add */
    ret = add3nums(server, n1, n2, n3, &result);
    if (ret != KERN_SUCCESS)
        printf("Call to add3nums failed.\n");
    else
        printf("According to the server, %d + %d + %d= %d.\n", n1,
            n2, n3, result);
}
```

Note that although the RPC looks like it directly calls **add2nums()** and **add3nums()** in the server, it really doesn't. The client instead sends a message that's received in **server_loop()**, which calls **add_server()**. The **add_server()** function calls **add2nums()** or **add3nums()** and passes the result back to the client in a message.

Making a function such as **add2nums()** an RPC gives the advantages of network independence, interface independence, and automatic type checking, at the expense of some complexity in the server.

MiG Specification File

You must first write a MiG specification file to specify the details of the procedure arguments and the messages to be used. A MiG specification file contains the following components, some of which may be omitted:

- Subsystem identification
- Type declarations
- Import declarations
- Operation descriptions
- Options declarations

The subsystem identification should appear first for clarity. Types must be declared before they're used. Code is generated for the operations and import declarations in the order in which they appear in the specification files. Options affect the operations that follow them.

See the earlier section, ªProgramming Example,º for a complete subsystem definition.

Subsystem Identification

The subsystem identification statement has the following form:

subsystem *sys message_base_id* ;

The *sys* is the name of the subsystem. It's used as the prefix for all generated file names. The user file name will be *sys***User.c**, the user header file will be *sys***.h**, and the server file will be *sys***Server.c**.

The *message_base_id* is a decimal integer that's used as the IPC message ID of the first operation in the specification file. Operations are numbered sequentially beginning with this base. The MiG-generated server function checks the message ID of an incoming message to make sure that it's no less than *message_base_id* and no greater than *message_base_id* + *num_messages* -1, where *num_messages* is the number of messages understood by the server.

Several servers can use just one message receive loop as long as they have different subsystem numbers (and they have few enough messages so that message IDs don't overlap). The message receive loop should call each MiG-generated server function in turn until one of them returns true (indicating the message ID is in the range understood by that server.) Once a MiG-generated server function has returned true or all the servers have returned false, the receive-serve-send loop should send a reply (unless the reply message returned by the server function has MIG_NO_REPLY in its **RetCode** field).

Example:

```
subsystem random 500;
```

Type Declarations

Simple Types

A simple type declaration has the following form:

type *user_type_name* = *type_desc* [*translation_info*]

where a *type_desc* is either a previously defined *user_type_name* or an *ipc_type_desc,* which has one of the following forms:

ipc_type_name
(*ipc_type_name* [*, size* [, **dealloc**]])

The *user_type_name* is the name of a C type that will be used for some parameters of the calls exported by the user interface file. The *ipc_type_desc* of simple types are enclosed in parentheses and consist of an IPC type name, decimal integer, or integer expression that's the number of bits in the IPC type and, optionally, the **dealloc** keyword.

The standard system-defined IPC type names are:

MSG_TYPE_BOOLEAN
MSG_TYPE_BIT
MSG_TYPE_BYTE
MSG_TYPE_CHAR
MSG_TYPE_INTEGER_8
MSG_TYPE_INTEGER_16
MSG_TYPE_INTEGER_32
MSG_TYPE_REAL
MSG_TYPE_STRING
MSG_TYPE_PORT
MSG_TYPE_PORT_ALL
MSG_TYPE_UNSTRUCTURED

The current set of these type names is contained in the header file **mach/message.h**, which defines all the message-related types needed by a user of the Mach kernel. The programmer may define additional types. If the *ipc_type_name* is a system-defined one other than MSG_TYPE_STRING, MSG_TYPE_UNSTRUCTURED, or MSG_TYPE_REAL, *size* (the bit length) need not be specified and the parentheses can be omitted.

The **dealloc** keyword controls the treatment of ports and pointers after the messages they're associated with have

been sent. The **dealloc** keyword causes the deallocation bit in the IPC message to be set on; otherwise, it's off. If **dealloc** is used with a port, the port is deallocated after the message is sent. If **dealloc** is used with a pointer, the memory that the pointer references will be deallocated after the message has been sent. An error results if **dealloc** is used with any argument other than a port or a pointer.

Some examples of simple type declarations are:

```
type int = MSG_TYPE_INTEGER_32;
type my_string = (MSG_TYPE_STRING,8*80);
type kern_return_t = int;
type disposable_port = (MSG_TYPE_PORT_ALL,32,dealloc);
```

The MiG-generated code assumes that the C types **my_string**, **kern_return_t**, and **disposable_port** are defined in a compatible way by a programmer-provided header file. The basic C and Mach types are defined in the header file **mach/std_types.defs**.

MiG assumes that any variable of type MSG_TYPE_STRING is declared as a C **char *** or **char** *array***[***n***]**. Thus it generates code for a parameter passed by reference and uses **strncpy()** for assignment statements.

Optional *translation_info* information describing procedures for translating or deallocating values of the type may appear after the type definition information:

- Translation functions, **intran** and **outtran**, allow the type as seen by the user process and the server process to be different.
- Destructor functions allow the server code to automatically deallocate input types after they have been used.

For example:

```
type task_t = (MSG_TYPE_PORT,32)
intran:    i_task_t PortToTask(task_t)
outtran:   task_t TaskToPort(i_task_t)
destructor: DeallocT(i_task_t)
;
```

Note: Because *translation_info* is part of the type declaration, the semicolon (**;**) doesn't appear until after the end of *translation_info*.

In this example, **task_t**, which is the type seen by the user code, is defined as a port in the message. The type seen by the server code is **i_task_t**, which is a data structure used by the server to store information about each task it's serving. The **intran** function **PortToTask()** translates values of type **task_t** to **i_task_t** on receipt by the server process. The **outtran** function **TaskToPort()** translates values of type **i_task_t** to type **task_t** before return. The destructor function **DeallocT()** is called on the translated input parameter, **i_task_type**, after the return from the server procedure and can be used to deallocate any or all parts of the internal variable. The destructor function won't be called if the parameter is also an **out** parameter (as described later in this chapter, in the section ªOperation Descriptionsº); this is because the correct time to deallocate an **out** parameter is after the reply message has been sent, which MiG doesn't do. A destructor function can also be used independently of the translation routines. For example, if a large out-of-line data segment is passed to the server, it could use a destructor function to deallocate the memory after the data was used.

Although calls to these functions are generated automatically by MiG, the function definitions must be hand-coded and imported using:

```
i_task_t PortToTask(task_t x)
task_t TaskToPort(i_task_t y)
void DeallocT(i_task_t y)
```

Structured Types

Three kinds of structured types are recognized: arrays, structures, and pointers. Definitions of arrays and structures have the following syntax:

array [*size*] **of** *comp_type_desc*

array [* : *maxsize*] **of** *comp_type_desc*
struct [*size*] **of** *comp_type_desc*

where *comp_type_desc* may be a simple *type_desc* or may be an **array** or **struct** type, and *size* may be a decimal integer constant or expression. The second array form specifies that a variable-length array is to be passed in-line in the message. In this form *maxsize* is the maximum length of the item. Currently, only one variable-length array may be passed per message. For variable-length arrays an additional count parameter is generated to specify how much of the array is actually being used.

If a type is declared as an **array**, the C type must also be an array, since the MiG RPC code will treat the user type as an array (that is, MiG will assume that the user type is passed by reference and it will generate special code for array assignments). A variable declared as a **struct** is assumed to be passed by value and treated as a C structure in assignment statements. There is no way to specify the fields of a C structure to MiG. The *size* and *type_desc* are used only to give the size of the structure. The following example shows how to declare a C structure as a **struct**.

```
/* declaration in MiG .defs file */
type short = MSG_TYPE_INTEGER_16;
type port_t = MSG_TYPE_PORT;
type lock_struct = struct [9] of short;
routine fl_message(server_port: port_t; inout arg: lock_struct);

/* declaration in C code */
typedef struct {
    short l_type;
    short l_whence;
    long l_start;
    long l_len;
    short l_pid;
    long l_hostid;
} lock_struct;
```

Pointer Types

In the definition of pointer types, the symbol ^ precedes a simple, array, or structure definition.

^ *comp_type_desc*
^ **array** [*size*] **of** *comp_type_desc*
^ **struct** [*size*] **of** *com_type_desc*

The *size* may be left blank or be *****. In either case, the array or structure is of variable size, and a parameter is defined immediately following the array parameter to contain its size. Data types declared as pointers are sent out-of-line in the message. Since sending out-of-line data is considerably more expensive than sending in-line data, pointer types should be used only for large or variable amounts of data. A call that returns an out-of-line item allocates the necessary space in the user's virtual memory. It's up to the user to call **vm_deallocate()** on this memory when finished with the data.

Some examples of complex types are:

```
type procids = array [10] of int;
type procidinfo = struct [5*10] of (MSG_TYPE_INTEGER_32);
type vardata = array [ * : 1024 ] of int;
type array_by_value = struct [1] of array [20] of (MSG_TYPE_CHAR);
type page_ptr = ^ array [4096] of (MSG_TYPE_INTEGER_32);
type var_array = ^ array [] of int;
```

Import Declarations

If any of the *user_type_name*s or *server_type_name*s are other than the standard C types (such as **int** and **char**), C type specification files must be imported into the user interface and server interface files so that they'll compile. The import declarations specify files that are imported into the modules generated by MiG.

An import declaration has one of the following forms:

import *file_name*;
uimport *file_name*;
simport *file_name*;

where *file_name* has the same form as file name specifications in **#include** statements (that is, **<***file_name***>** or **"***file_name***"**).

For example:

```
import "my_defs.h";
import "/usr/include/mach/cthreads.h";
import <mach/cthreads.h>;
```

Files included with **import** are included in both the user-side and server-side code. Those included with **uimport** are included in just the user side. Those included with **simport** are included in just the server side.

Operation Descriptions

Any of five standard operations may be specified by using the following keywords:

function
routine
procedure
simpleprocedure
simpleroutine

One other keyword, **skip**, may be used in place of a standard operation.

Functions and routines have a return value; procedures don't. Routines are functions whose result is of type **kern_return_t**. This result indicates whether the requested operation was successfully completed. If a routine returns a value other than KERN_SUCCESS, the reply message won't include any of the reply parameters except the error code. Neither procedures nor functions return indications of errors directly; instead they call a hand-coded error function in the client. The name of the error function is **MsgError()**, by default; you can specify another name using the **error** declaration in the MiG specification file.

Simple procedures and simple routines send a message to the server but don't expect a reply. The return value of a simple routine is the value returned by the function **msg_send()**. Simple routines or simple procedures are used when asynchronous communication with a server is desired. The rest of the operations wait for a reply before returning to the caller.

The syntax of the **procedure**, **simpleprocedure**, **simpleroutine**, and **routine** statements are identical. The syntax of **function** is also the same except for the type name of the value of the function. The general syntax of an operation definition for everything except **function** has the following form:

operation_type operation_name (*parameter_list*) ;

For **function** the form is:

function *operation_name* (*parameter_list*) : *function_value_type* ;

The *parameter_list* is a list of parameter names and types separated by a semicolon. The form of each parameter is:

[*specification*] *var_name* : *type_description* [, **dealloc**]

If not omitted, *specification* must be one of the following:

in
out
inout
requestport
replyport

waittime
sendtime
msgtype

The *type_description* can be any *user_type_name* or a complete type description (see the earlier section in this chapter, ªType Declarationsº).

The first unspecified parameter in any operation statement is assumed be the **requestport** unless a **requestport** parameter was already specified. This is the port that the message is to be sent to. If a **replyport** parameter is specified, it will be used as the port that the reply message is sent to. If no **replyport** parameter is specified, a per-thread global port is used for the reply message.

The keywords **in**, **out**, and **inout** are optional and indicate the direction of the parameter. The keyword **in** is used with parameters that are to be sent to the server. The keyword **out** is used with parameters to be returned by the server. The keyword **inout** is used with parameters to be both sent and returned. If no such keyword is given, the default is **in**.

The keywords **waittime**, **replyport**, and **msgtype** can be used to specify dynamic values for the wait time, the reply port, or the message type for this message. These parameters aren't passed to the server code, but are used when generating the send and receive calls. The **requestport** and **replyport** parameters must be of types that resolve to MSG_TYPE_PORT. The **waittime** and **msgtype** parameters must resolve to MSG_TYPE_INTEGER_32.

The keyword **skip** is provided to allow a procedure to be removed from a subsystem without causing all the subsequent message interfaces to be renumbered. It causes no code to be generated, but uses up a **msg_id** number.

Here are some examples:

```
procedure init_seed (       server_port : port_t;
                            seed        : dbl);
routine get_random (        server_port : port_t;
                            out num     : int);
simpleroutine use_random (  server_port : port_t;
                            info_seed   : string80;
                            info        : comp_arr;
                            info_1      : words);
simpleprocedure exit (      server_port : port_t);
```

See the earlier section in this chapter, ªProgramming Example,º for an example of a complete subsystem definition.

Options Declarations

Several special-purpose options about the generated code may be specified. Defaults are available for each, and simple interfaces don't usually need to change them. First-time readers may want to skip this section. These options may occur more than once in the specification file. Each time an option declaration appears, it sets that option for all the following operations.

The waittime Specification

The **waittime** specification has one of the following two forms:

waittime *time* **;**
nowaittime ;

The word **waittime** is followed by an integer or an identifier that specifies the maximum time in milliseconds that the user code will wait for a reply from the server. If an identifier is used, it should be declared as an **extern** variable by some module in the user code. If the **waittime** option is omitted, or if the **nowaittime** statement is seen, the RPC doesn't return until a message is received.

The timeout value for the **msg_receive()** can alternatively be controlled by using a **waittime** parameter to the RPC.

The sendtime Specification

The **sendtime** specification has one of the following two forms:

sendtime *time* **;**
nosendtime ;

The word **sendtime** is followed by an integer or an identifier that specifies the maximum time in milliseconds that the user code will wait for the number of messages queued on the server's port to fall below the port's backlog. If an identifier is used, it should be declared as an **extern** variable by some module in the user code. If the **sendtime** option is omitted, or if the **nosendtime** statement is seen, the RPC doesn't return until the message has been enqueued on the server's port.

The timeout value for the **msg_send()** can be controlled alternatively by using a **sendtime** parameter to the RPC.

The msgtype Specification

The **msgtype** specification has the following form:

msgtype *msgtype_value* **;**

msgtype_value may be one of the values from the header file **mach/msg_type.h**. The available types are MSG_TYPE_RPC and MSG_TYPE_NORMAL. The MSG_TYPE_RPC is set to a correct value by default; this value normally shouldn't be changed. The value MSG_TYPE_NORMAL can be used to reset the **msgtype** option.

The **msgtype** value for the **msg_send()** can be controlled alternatively by using a **msgtype** parameter to the RPC.

The error Specification

The **error** specification has the following form:

error *error_proc* **;**

The **error** specification is used to specify how message-passing errors are to be handled for operations other than routines or simple routines. In all types of routines, any message errors are returned in the return value of the routine. For operations of types other than routines, the procedure *error_proc* is called when a message error is detected. The procedure specified by *error_proc* has to be supplied by the user, and must be of the form:

void *error_proc* (kern_return_t *error_code*)

If the **error** specification is omitted, *error_proc* is set to **MsgError()**.

The serverprefix Specification

The **serverprefix** specification has the following form:

serverprefix *string* **;**

The word **serverprefix** is followed by an identifier string that will be prepended to the actual names of all the following server-side functions implementing the message operations. This is particularly useful when it's necessary for the user-side and server-side functions to have different names, as must be the case when a server is also a user of copies of itself.

The userprefix Specification

The **userprefix** specification has the following form:

userprefix *string* **;**

The word **userprefix** is followed by an identifier string that will be prepended to the actual names of all the following user-side functions calling the message operations. **serverprefix** should usually be used when different names are needed for the user and server functions, but **userprefix** is also available for the sake of completeness.

The rcsid Specification

The **rcsid** specification has the following form:

rcsid *string* **;**

This specification causes a string variable *sys*_**user_rscid** in the user module and *sys*_**server_rcsid** in the server module to be set equal to the input string. The subsystem name *sys* was described earlier in this chapter, in the section ªSubsystem Identification.º

Syntax Summary

This section summarizes the syntax of MiG specification files. Note the following conventions:

- Terminal symbols (literals) are shown in boldface type.
- Nonterminal symbols are shown in italic type.
- Alternatives are listed on separate lines.
- Brackets indicate zero or one occurrence of the bracketed item. An ellipsis (...) indicates one or more repetitions of the preceding item. Brackets and ellipsis combined, as in [€*item* ...] indicate zero, one, or more repetitions of the item.
- Types must be declared before they're used.
- Comments may be included in a ª.defsº file if surrounded by **/*** and ***/**. Comments are parsed and removed by the C preprocessor.

specification_file:
subsystem_description [*waittime_description*] [*sendtime_description*]
[*msgtype_description*] [*error_description*] [*server_prefix_description*]
[*user_prefix_description*] [*rscid_description*] [*type_description ...*]
[*import_declaration ...*] *operation_description ...*

subsystem_description:
subsystem *identifier decimal_integer* **;**

waittime_description:
waittime *time_value* **;**
nowaittime ;

sendtime_description:
sendtime *time_value* **;**
nosendtime ;

time_value:
MSG_TYPE_INTEGER_32

msgtype_description:
msgtype *msgtype_value* **;**

msgtype_value:
MSG_TYPE_RPC
MSG_TYPE_NORMAL

error_description:
error *error_procedure* **;**

server_prefix_description:

serverprefix *identifier_string* **;**

user_prefix_description:
userprefix *identifier_string* **;**

rcsid_description:
rcsid *identifier_string* **;**

type_description:
type *type_definition* **;**

import_declaration:
import_keyword include_name **;**

import_keyword:
import
uimport
simport

include_name:
"*file_name***"**
<*file_name***>**

operation_description:
routine_description
simpleroutine_description
procedure_description
simpleprocedure_description
function_description

routine_description:
routine *argument_list* **;**

simpleroutine_description:
simpleroutine *argument_list* **;**

procedure_description:
procedure *argument_list* **;**

simpleprocedure_description:
simpleprocedure *argument_list* **;**

function_description:
function *argument_list* **:** *type_definition* **;**

argument_list:
([*argument_definition*] [**;** *argument_definition*] ... **)**

argument_definition:
[*specification*] *identifier* **:** *type_definition* [**, dealloc**]

specification:
in
out
inout
requestport
replyport
waittime
msgtype

type_definition:
identifier **=** [^] [*repetition ...*] *ipc_info* [*translation*]

repetition:
    array [[*size*] **] of**
    struct [[*size*] **] of**

size:
    integer_expression

integer_expression:
    integer_expression **+** *integer_expression*
    integer_expression **-** *integer_expression*
    integer_expression ***** *integer_expression*
    integer_expression **/** *integer_expression*
    (*integer_expression* **)**
    integer

ipc_info:
    (*ipc_type_name* **,** *size_in_bits* [**, dealloc**] **)**
    ipc_type_name
    identifier

translation:
    [*input_function*] [*output_function*] [*destructor_function*]

input_function:
    intran : *identifier*

output_function:
    outtran : *identifier*

destructor_function:
    destructor : *identifier*

ipc_type_name:
    integer
    manifest_constant

Compiling MiG Specification Files

To compile a MiG specification file, specify the name of your ªdefsº file (or files) and any switches as arguments to the **mig** command. For example:

```
mig -v random.defs
```

MiG recognizes the following switches:

[**handler** *name*]
Specifies a name for the file that's usually called *sys***Server.c**, and specifies that it should provide a handler interface instead of the usual server interface. An additional header file called *sys***_handler.h** is also produced, as if the **sheader** option were specified. Handler interfaces are used mainly in loadable kernel servers; they're discussed in Chapter 6, ªDesigning Loadable Kernel Servers.º

[**header** *name*]
Specifies a name for the file that's usually called *sys***.h**.

[**p**,**P**] If **p**, use 2-byte message padding. You should use this option only if your server or client might be exchanging messages containing fields shorter than 4 bytes with a client or server that was built using NeXT Software Release 1. If **P**, use 4-byte message padding. The default value is **P**. For example, a 1-byte message element would be padded to 2 bytes if you specify **p**, or 4 bytes by

default.

[**q**,**Q**] If **q**, suppress warning statements. If **Q**, print warning statements. The default value is **Q**.

[**r**,**R**] If **r**, use **msg_rpc()**; if **R**, use **msg_send()**, **msg_receive()** pairs. The default value is **r**.

[**s**,**S**] If **s**, generate symbol table with *sys***Server.c** code. The layout of a symbol table (**mig_symtab_t**) is defined in the header file **mach/mig_errors.h**. If **S**, suppress the symbol table. The default value is **S**. This is useful for protection systems where access to the server's operations is dynamically specifiable or for providing a run-time indirected server call interface with **syscall()** (server-to-server calls made on behalf of a client).

[**server** *name*]
Specifies a name for the file that's usually called *sys***Server.c**.

[**sheader** *name*]
Specifies that MiG create an additional header file, called *name*, that's suitable for inclusion in the server defined by the ª.defsº file.

[**user** *name*]
Specifies a name for the file that's usually called *sys***User.c**.

[**v**,**V**] If **v** (verbose), print routines and types as they're processed. If **V**, compile silently. The default value is **V**.

Any switches MiG doesn't recognize are passed to the C preprocessor. MiG also notices if the **-MD** option is being passed to the C preprocessor. If it is, MiG fixes up the resulting ª.dº file to show the dependencies of the ª.h,º and ª.cº files on the ª.defsº file and any included ª.defsº files. For this feature to work correctly, the name of the subsystem must be the same as the name of the ª.defsº file.

MiG runs the C preprocessor to process comments and preprocessor macros such as **#include** or **#define**. For example, the following statement can be used to include the type definitions for standard Mach and C types:

```
#include <mach/std_types.defs>
```

The output from the C preprocessor is then passed to the program **migcom**, which generates the C files.