MIG - The MACH Interface Generator

Richard P. Draves
Michael B. Jones
Mary R. Thompson

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Version of:
2 November 1989

Abstract

Matchmaker is a language for specifying and automating the generation of multi-lingual interprocess communication interfaces. MIG is an interim implementation of a subset of the Matchmaker language that generates C and C++ remote procedure call interfaces for interprocess communication between MACH tasks.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-84-C-0467.

1. Introduction

MIG is a program which generates remote procedure call (RPC) code for communication between a client and a server process. MACH servers execute as separate tasks and communicate with their clients by sending MACH inter-process communication (IPC) messages. The IPC interface is language independent and fairly complex. The MIG program is designed to automatically generate procedures in C to pack and send, or receive and unpack the IPC messages used to communicate between processes.

The user must provide a specification file defining parameters of both the message passing interface and the procedure call interface. MIG then generates three files:

- **User Interface Module**: This module is meant to be linked into the client program. It implements and exports procedures and functions to send and receive the appropriate messages to and from the server.
- **User Header Module**: This module is meant to be included in the client code to define the types and routines needed at compilation time.
- **Server Interface Module**: This module is 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 generated interface module gathers the output parameters and correctly formats a reply message.

2. MIG Specification File

A MIG specification file contains the following components, some of which may be omitted.

1. Subsystem identification
2. Type specifications
3. Import declarations
4. Operation descriptions
5. Options declarations

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

2.1. Subsystem Identification

The subsystem identification statement is of the form

`subsystem` *sys message-base-id* `;`

sys is the name of the subsystem. It is 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`. These file names may be overridden by command line switches.

message-base-id is a decimal integer that is used as the IPC message ID of the first operation in the specification file. Operations are numbered sequentially beginning with this base. The *message-base-id* can be selected arbitrarily by the implementor of the subsystem, but it is recommended for "official" subsystem implementors to request an unused *message-base-id* from either the MACH librarian or the Camelot librarian. The only technical requirement is that all requests sent to a single service port should have different IDs, to allow the server to easily distinguish them.

2.2. Type Declarations

2.2.1. Simple Types

A simple type declaration is of the form:

`type` *user-typename* = *type-desc* [translation-info]

where a *type-desc* is either a previously defined *user-typename* or an *ipc-type-desc* which is of the form:

`(`*ipc-typename*`,`*size* `[ ,` *dealloc-flag* `])` or
ipc-typename

The *user-typename* is the name of a C type that will be used for some parameters of the calls exported by the User Interface Module. The *ipc-type-desc* of simple types are enclosed in parentheses and consist of an IPC typename, decimal integer or integer expression that is the number of bits in the IPC type, and an optional deallocation keyword which is either `dealloc` or `notdealloc`. The currently defined standard IPC typenames are given in Appendix II. The user may define additional types. If the *ipc_typename* is one of the standard ones, except for `MSG_TYPE_STRING`, `MSG_TYPE_UNSTRUCTURED` or `MSG_TYPE_REAL`, just the *ipc_typename* can be used.

The deallocation keyword is used to describe the treatment of ports and pointers after the messages with which they are associated have been sent. `dealloc` causes the deallocation bit in the IPC message to be set on. Otherwise it is always off. `dealloc` can be used only with ports and pointers. If it is used with a port, the port will be 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 will result 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 user provided include file. The files `<mach/std_types.defs>` and `<mach/mach_types.defs>` define the basic C and MACH types.

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

2.2.2. Structured Types

Three kinds of structured types are recognized: arrays, structures and pointers. Definitions of arrays and structures follow the Pascal-style syntax of:

`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. For variable length arrays an additional count parameter is generated to specify how much of the array is actually being used. Variable-length inline InOut arguments are not supported.

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 (i.e. asssuming it is passed by reference and generating 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 just used to give the size of the structure.

2.2.3. 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` *comp-type-desc*

In this case `size` may be left blank or be `*`. In either of these cases, 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 is considerably more expensive than in-line data, pointer types should only be used 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 is up to the user to deallocate this memory when he is 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;
```

2.2.4. Polymorphic Types

MIG supports polymorphic types. For example, using this facility, one may specify an argument which can be either an integer or a port, with the exact type determined at runtime. The type information is passed in an auxiliary argument, similar to the way size information in variable-sized arrays is handled. (If an argument is both variable-sized and polymorphic, the auxiliary type argument comes before the count argument.)

```
type poly_t = polymorphic;

simpleroutine SendPortOrInt(
                server  : port_t;
                poly    : poly_t);
```

and then in client C code

```
port_t server, port;
kern_return_t kr;

kr = SendPortOrInt(server, 5, MSG_TYPE_INTEGER_32);
kr = SendPortOrInt(server, port, MSG_TYPE_PORT);
```

MIG also supports types which change during transmission. The syntax is

```
type int2port = MSG_TYPE_INTEGER_32 | MSG_TYPE_PORT
        CUserType: int
        CServerType: port_t;

simpleroutine SendInt2Port(
                server  : port_t;
                arg     : int2port);
```

This functionality is mainly useful in conjunction with the MSG_TYPE_INTERNAL_MEMORY type inside the kernel.

2.2.5. Type translation information

Optional information describing procedures for translating or deallocating values of the type may appear after the type definition information. Translation functions 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. An example is:

```
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);
```

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 some data structure used by the server to store information about each task it is 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 will not be called if the parameter is also an out argument, because the correct time to deallocate an out parameter is after the reply message has been sent, which is not code that is generated by MIG. A destructor function can also be used independly 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 dellocate 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(x)
        task_t  x;

task_t TaskToPort(y)
        i_task_t  y;

void  DeallocT(y)
        i_task_t  y;
```

For each MIG type, there are up to three corresponding C types. These are the type used in the client module, the type used in the server module, and the translated type used by the server procedure. By default these three types are all the same, and have the same name as the MIG type. The `InTran` and `OutTran` options implicitly specify the server type and translated type names. The `Destructor` option implicitly specifies the translated type name. The user type and server type may be specified with the `CUserType` and `CServerType` options, or both specified together with the `CType` option. For example,

```
type dealloc_port_t = (MSG_TYPE_PORT, 32, dealloc)
        CType: port_t;

type funny_int = int
        CUserType: user_int
        CServerType: server_int;
```

2.3. Import Declarations

If any of the user-typenames or server-typenames are other than the standard C types (e.g. `int`, `char`, etc.) C type definition files must be imported into the User and Server Interface modules so that they will compile. The import declarations specify files which are imported into the modules generated by MIG.

An import declaration is of one of the following forms:

`import` *file-name*;
`uimport` *file-name*;
`simport` *file-name*;

where *file-name* is in a form acceptable by `cpp` in `#include` statments, e.g. `<file-name>` or `"file-name"`.

For example:

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

`import` declarations are included in both the user and server side code. `uimport` declarations are included in just the user side. `simport` declarations are included in just the server side.

2.4. Standard Operations

Two kinds of standard operations may be specified:

1. `Routine`
2. `SimpleRoutine`

3. `Procedure`
4. `SimpleProcedure`
5. `Function`

One other keyword may be used in place of a standard operation and that is `skip`. It produces a hole in the IPC message ID numbering of operations.

A `SimpleRoutine` sends a message to the server but does not expect a reply. The return value of a `SimpleRoutine` is the value returned by the `msg_send` primitive. A `SimpleRoutine` is used when asynchronous communication with a server is desired.

A `Routine` operation waits for a reply message from the server. It is a function 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 will not include any of the reply parameters except the error code.

`Procedure` operations are like `Routines`, except that they don't return an error code. `SimpleProcedures` resemble `SimpleRoutines`, with the same exception. `Function` operations also don't return an error code; instead, they return a value from the server function. This value can be any scalar or structure type. The syntax is

`function` *operation-name* `(` *parameter-list* `)` `:` *function-value-type* `;`

Because `Procedure`, `SimpleProcedure`, and `Function` operations don't return an error code, they have another way of handling errors. They call an error function, specified with an `error` option. These calls are not recommended since it is tempting to write code that ignores message passing errors with these operations and that is definitely a bad idea.

An operation definition has the syntax

operation-type *operation-name* `(` *parameter-list* `);`

The *parameter-list* is a list of parameter names and types separated by `;`. The form of each parameter is:

specification *var-name* `:` *type-description* `[` `,`*dealloc-flag* `]`

where *specification* may either be omitted or be one of the following `in` | `out` | `inout` | `RequestPort` | `ReplyPort` | `WaitTime` | `MsgType`. *Type-description* can be any *user-typename* that was declared in the type definitions section or can be a complete type description in the same form as in the type definition section. *dealloc-flag* may be omitted or be `dealloc` or `notdealloc`. This flag will overide any deallocation flag associated with the type definition.

The first unspecified parameter in any operation statement is assumed be the `RequestPort` unless a `RequestPort` parameter was already specified. This is the port to which the message is to be sent. 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. If no such keyword is given the default is `in`. 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. The keywords `WaitTime, ReplyPort` and `MsgType` may be used to specify dynamic values for the `WaitTime`, the `ReplyPort` or the `MsgType` for this message. These parameters are not 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` and the `WaitTime` and `MsgType` parameters must resolve to `MSG_TYPE_INTEGER_32`.

The keyword `skip` is provided to allowed 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.

See appendix III for an example of a complete subsystem definition.

2.5. Options Declarations

Several special-purpose options about the generated code may be specified. Defaults are available for each, and simple interfaces do not usually need to change them. First time readers may wish 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.

2.5.1. WaitTime Specification

The `WaitTime` specification has two forms:

> `WaitTime` *time* `;` or
> `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 does not 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.

2.5.2. MsgType Specification

The `MsgType` specification is of the form

> `MsgType` *manifest-constant* `;`

where the *manifest-constant* may be one of the values from the file <msg_type.h>. The currently available types are `MSG_TYPE_RPC` and `MSG_TYPE_ENCRYPTED`. The type `MSG_TYPE_CAMELOT` should not be set in this manner, but is set by using the operation `camelotroutine`. The `MSG_TYPE_RPC` is set to a correct value by default and the user should probably not change it. The value `MSG_TYPE_NORMAL` can be used to reset the `MsgType` option.

The `MsgType` value for the `msg_send` can alternatively be controlled by using a `MsgType` parameter to the RPC.

2.5.3. Error Specification

The `error` specification required by `Procedure`, `SimpleProcedure`, and `Function` operations is of the form

```
error error-proc;
```

The *error-proc* has to be supplied by the user and must be of the form:

```
void error-proc (ErrCode)
        kern_return_t   ErrCode;
```

If there is no `error` specification, the default *error-proc* is `MsgError`.

2.5.4. ServerPrefix Specification

The ServerPrefix specification is of the form

```
ServerPrefix string ;
```

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

2.5.5. UserPrefix Specification

The UserPrefix specification is of the form

```
UserPrefix string ;
```

The word `UserPrefix` is followed by an identifier string which 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 completeness sake.

2.5.6. RcsId Specification

The RcsId specification is of the form

```
rcsid "$Header information$";
```

This specification causes a string variable *Sys*`_user_rscid` in the user and *Sys*`_server_rcsid` in the server module to be set equal to the input string.

3. Obsolete Functionality

MIG supports some obsolete functionality. Use of this functionality will result in a warning message, unless warning messages are suppressed. This functionality may disappear in a future release.

3.1. Kernel Subsystems

The keyword `kernel` was once used for interfaces supported by the kernel.

```
subsystem kernel sys message-base-id ;
```

3.2. Old Translation Syntax

The previous syntax for specifying translation and destructor functions for a type was based on key characters. The `task_t`/`i_task_t` example would be rendered

```
type task_t = MSG_TYPE_PORT
   > PortToTask
   < TaskToPort
   : i_task_t
   ~ task_t
   - DeallocT;
```

4. Camelot Functionality

MIG supports the Camelot system with an additional type of operation, the `CamelotRoutine`. Camelot interfaces should be entirely composed of `CamelotRoutines`; they shouldn't be mixed with other operations. In addition, the `subsystem` declaration of a Camelot interface should include the `camelot` keyword:

`subsystem camelot` *sys message-base-id* `;`

See the Camelot documentation *Guide to the Camelot Distributed Transaction Facility* for a full description of how `CamelotRoutines` differ from `Routines`.

5. Compiling Definitions Files

MIG is implemented as a cover program that recognizes a few switches and then calls `cpp` to process comments and preprocessor macros such as `#include` or `#define`. For example the statement

#include <std_types.defs>

can be used to include the type definitions for standard MACH and C types. The output from `cpp` is then passed to the program `migcom` which generates the C files.

The switches that MIG recognizes are:

-[r,R]	r use msg_rpc, R use msg_send, msg_receive pairs. Default is r.
-[q,Q]	q - suppress warning statements. Q print warning statements. Default is Q.
-[v,V]	v - verbose, prints out routines and types as they are processed. V compiles silently. Default is V.
-[s,S]	s - generate symbol table with Server.c code. The layout of a symbol table (`mig_symtab_t`) is defined in <`mig_error.h`>. S suppresses the symbol table. Default 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 ala `syscall(2)` (server-to-server calls made on behalf on a client).
-i	- instead of a single user file, generate individual files for each routine, for ease in building a library. The file name for each file is *routine_name*.c.
-server *name*	- name the server file *name*.
-user *name*	- name the user file *name*.
-header *name*	- name the header file *name*.

Any switches that MIG does not recognize, it passes on to `cpp`. MIG also notices if -MD is being passed to cpp. If it is, MIG fixes up the resulting .d file to show the dependencies of the .h, User.c and Server.c on the .defs and any #included .defs files. For this feature to work correctly the name of the subsystem must be the same as the name of the .defs file.

To use MIG, give the name of your .defs file or files and any switch values on a MIG command line, for example:

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

If random is the subsystem name declared in the definitions file, then MIG will produce the files random.h, randomUser.c and randomServer.c as output. If the -MD switch was given, a random.d file will also be generated.

6. Using the Interface Modules

In the following discussion let random be the declared subsystem name in the definitions file.

To use the calls exported by the user interface module, a client must first find the port to call the server on. The service ports for basic system servers are inherited when a task is created and can be found in globals defined by mach_init. One of these ports is the port to the Net Name Server which can be used to check in or lookup user supplied ports. The ReplyPort for MIG is a per-thread global that is intialized when the thread is created. The program set_mig_port can be called to reset this value, but it is not recommended to do so. The ReplyPort can be specified as a parameter to an operation if necessary.

If the interface calls an error handling routine the user needs to supply a properly named routine, either MsgError, or whatever the interface writer specified as the error_proc.

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

The most common system error that a user of MIG interface may encounter is invalid_port. This can mean serveral things: 1. The RequestPort parameter is an invalid port or lacks send rights. 2. The ReplyPort is invalid or lacks receive rights. If the client is supplying this port as parameter it may be at fault. If the system provided reply port is being used this error should not happen. 3. A port that is being passed in either the send or reply message is invalid. timed_out is another system error a client could receive. This results from a RPC with a timeout value set, timing out before a reply is received. This usually only happens if the server is on a remote machine from the client. The MIG errors defined in mig_errors.h usually only occur if the client is using a different version of the interface than the server. MIG error codes can be interpreted by the routines in mach_error

The subsystem writer must hand code two things in addition to the MIG definition file. First, the actual operations must be declared and imported into the server module, and all normal operations must be coded. Second, code must be written to receive messages, call the server interface module, and then send a reply message when appropriate. The server module exports one function called random_server, which accepts as arguments a pointer to the message received, and a pointer to a record for the reply message. The function will return true if the received message id was in the server's range.

In general, a reply should always be returned for any message received unless the return code from the Server was `MIG_NO_REPLY` or the request message doesn't have a reply port. The boolean function value from the server function may be used to have the same receive loop processing several logically distinct server's requests. Once a server has returned true, or all the servers have returned false the receive-serve-send loop should send a reply (unless of course, the return code was `MIG_NO_REPLY` or the reply port is PORT_NULL).

An example of a server main loop can be found in Appendix III.

I. Syntax of the Specification file

The syntax of specification files is given semi-formally in what can be viewed as a slightly extended context free grammar.

Meta language description: Each production rule is terminated with a period. The left hand side is separated from the right hand side by a colon. Terminal symbols are enclosed in double quotes. Nonterminal symbols are not specially marked and may contain embeded blanks. Symbols within a single production are separated by commas. Alternative productions are separated by '|'. A nonterminal ending with the word 'option' indicates zero or one occurance of the nonterminal with that word deleted. A nonterminal ending with the word 'sequence' indicates one or more repetitions of the nonterminal with that word deleted. (It follows that a nonterminal ending with 'sequence option' indicates zero or more repetitions.)

Comments may be included in .defs file if surrounded by "/*" and "*/". They are parsed and removed by cpp.

No attempt has been made to indicate in the grammar that types must be declared before they are used.

```
specification file:
        subsystem description,
        waittime description option,
        msgtype description option,
        error description option,
        server prefix description option,
        user prefix description option,
        rscid description option
        type description sequence option,
        import statement sequence option,
        operation description sequence.

subsystem description:
        "Subsystem", identifier, decimal integer, ";".

waittime description:
        "WaitTime", waittime value, ";" |
        "NoWaitTime",";".

waittime value:
        decimal integer |
        identifier.

msgtype description:
        "MsgType", msgtype value, ";".

msgtype value:
        "MSG_TYPE_RPC" |
        "MSG_TYPE_ENCRYPTED" |
        "MSG_TYPE_NORMAL" .

error description:
        "Error", identifier, ";".

server prefix description:
        "ServerPrefix", identifier ";".
```

```
user prefix description:
        "UserPrefix", identifier, ";".

import statement:
        import indicant, include name, ";".

include name:
        "file_name"|
        <file_name>.

import indicant:
        "import"|
        "uimport"|
        "simport".

type description:
        "type", type definition, ";".

operation description:
        routine description |
        simpleroutine description |
        procedure description |
        simpleprocedure description |
        function description |
        camelotroutine 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, ";".

camelotroutine description:
        "routine", argument list, ";".


argument list:
        "(", argument definition option,
                semicolon joined argument definition sequence option, ")".

semicolon joined argument definition:
        ";", argument definition.

argument definition:
        specification option, identifer, ":", type definition,
        ipc flag sequence option.

specification:
```

```
        "in" |
        "out" |
        "inout" |
        "requestport" |
        "replyport" |
        "waittime" |
        "msgtype".

type definition:
        identifier, "=",
                pointer option, repetition option sequence,
                ipc_info, translation option.

pointer: "^".

repetition:
        "array", "[", size option, "]", "of"|
        "struct", "[", size option, "]", "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 expression, ")" |
                        integer.

ipc_info:
        "(", ipc_type_name, ",", size in bits,
                ipc flag sequence option, ")" |
        ipc_type_name |
        identifier.

ipc_type_name:
        prim_ipc_type_name |
        prim_ipc_type_name, "|", prim_ipc_type_name.

prim_ipc_type_name:
        "polymorphic" |
        builtin type name |
        integer expression.

translation:
        ctype option, cusertype option, cservertype option,
        input function option, output function option,
        destructor function option.

ctype:
        "CType", ":", identifier.

cusertype:
        "CUserType", ":", identifier.
```

```
cservertype:
        "CServerType", ":", identifier.

input function:
        "InTran", ":", identifier, identifier, "(", indentifier, ")".

output function:
        "OutTran", ":", identifier, identifier, "(", indentifier, ")".

destructor function:
        "Destructor", ":", identifier, "(", indentifier, ")".

ipc flag:
        ",", flag identifier.

flag identifer:
        "dealloc" |
        "notdealloc" |
        "islong" |
        "isnotlong".
```

II. System defined IPC types

The builtin IPC typenames defined by the system are as follows:

```
MSG_TYPE_UNSTRUCTURED
MSG_TYPE_BIT
MSG_TYPE_BOOLEAN
MSG_TYPE_INTEGER_16
MSG_TYPE_INTEGER_32
MSG_TYPE_PORT_OWNERSHIP
MSG_TYPE_PORT_RECEIVE
MSG_TYPE_PORT_ALL
MSG_TYPE_PORT
MSG_TYPE_CHAR
MSG_TYPE_BYTE
MSG_TYPE_INTEGER_8
MSG_TYPE_REAL
MSG_TYPE_STRING
MSG_TYPE_PORT_NAME
MSG_TYPE_INTERNAL_MEMORY
MSG_TYPE_POLYMORPHIC
```

The most up-to-date set of these names is to be found in the C file `<mach/message.h>` which defines all the message related types needed by a user of the MACH kernel.

III. Example of MIG specification file

Here is an example of a simple MIG specification file. This code and the mig generated files `random.h`, `randomUser.c` and `randomServer.c` can all be found in the directory `/afs/cs.cmu.edu/project/mach/doc/mig_example`.

```
/***************************************************************
 *  random.defs
 *      Defines a  server to return random numbers, random strings
 *      and pages of memory filled with random junk.
 **************************************************************/
subsystem random 500;
#include <mach/std_types.defs>

type dbl        = struct [2] of int;
type string25   = (MSG_TYPE_STRING,8*25); /* passed by reference */
type string80   = struct [1] of array [80] of char; /* passed by value */
type words      = ^ array [ ] of int;
type words256   = array [256] of int;
type comp_arr   = array [10] of array [256] of int;
type page_ptr   = ^array [4096] of MSG_TYPE_INTEGER_32;

import "random_types.h";

procedure init_seed (   server_port     : port_t;
                        seed            : dbl);

function get_randomf(   server_port     : port_t)
        : int;

routine get_random(     server_port     : port_t;
                    out num             : int);

msgtype MSG_TYPE_ENCRYPTED;  /* get secret string */
waittime 10000;  /* next call may go to a server on another machine */

routine get_secret(     server_port     : port_t;
                  inout password        : string25);

/* input waittime to next procedure in case 10 seconds is not long enough */
/* mtype can be input as MSG_TYPE_NORMAL if encryption is  not wanted */

routine get_confidential(server_port    : port_t;
                waittime wait           : int;
                 MsgType mtype          : int;
                     out data           : page_ptr,dealloc);

nowaittime;
msgtype MSG_TYPE_NORMAL;

simpleroutine use_random(server_port    : port_t;
                        info_seed       : string80;
                        info            : comp_arr;
                        info_1          : words);

/* tell server you are finished using it */
simpleprocedure  exit(  server_port     : port_t);
```

`std_types.defs` defines the following types:

```
/************************************************************
 *      Mach kernel standard interface type declarations
 ************************************************************/

type char = MSG_TYPE_CHAR;
type short = MSG_TYPE_INTEGER_16;
type int = MSG_TYPE_INTEGER_32;
type boolean_t = MSG_TYPE_BOOLEAN;

type kern_return_t = int;

/* Until MIG & netmsgserver are updated, use integer type */
#define MSG_TYPE_PORT_NAME      MSG_TYPE_INTEGER_32

type port_name_t = MSG_TYPE_PORT_NAME;
type port_name_array_t = ^array[] of port_name_t;
type port_type_t = int;
type port_type_array_t = ^array[] of port_type_t;
type port_set_name_t = port_name_t;

type port_t = MSG_TYPE_PORT;
type port_all_t = MSG_TYPE_PORT_ALL;
type port_rcv_t = MSG_TYPE_PORT_RECEIVE;
type port_array_t = ^array[] of port_t;

type pointer_t = ^array [] of MSG_TYPE_BYTE;

import <mach/std_types.h>;
```

`random_types.h` defines the C types appropriately:

```
/*********************************************************
 *  random_types.h
 *      definitions for the random server interface
 *********************************************************/

typedef struct {
            int lsw;
            int msw;
        } dbl;

typedef struct {                /* gets passed by value */
            char  c[80];
        } string80;

typedef char    string25[25];   /* gets passed by reference */
typedef int     words256[256];
typedef int     *words;         /* variable length array */
typedef words256 comp_arr[10];
typedef int     pagearr[4096];
typedef pagearr *page_ptr;
```

The following code fragment shows a typical main loop for a server process.

```
/**************************************************
 *   Main program for random server
 *************************************************/
#include <stdio.h>
#include <mach.h>
#include <mach_error.h>
#include <mig_errors.h>
#include <mach/message.h>
#include <mach/notify.h>
#include <servers/envmgr.h>

extern boolean_t        random_server();

/********************************************************
 * procedure random_run:
 *   Waits for messages to server,
 *   handles them, and replies to sender.
 *******************************************************/
void random_run ()
{
   typedef int space[1024]; /* Maximum message size */

   typedef struct DumMsg
        {
           msg_header_t         head;
           msg_type_t           retcodetype;
           kern_return_t        return_code;
           space                body;
        } DumMsg;

   kern_return_t        retcode;
   msg_return_t         msgcode;
   boolean_t            ok;
   DumMsg               *pInMsg, *pRepMsg;

   pInMsg = (DumMsg *)malloc(sizeof(DumMsg));
   pRepMsg = (DumMsg *)malloc(sizeof(DumMsg));

   while (TRUE)
   {
       pInMsg->head.msg_size = sizeof(DumMsg); /* bytes */
       pInMsg->head.msg_local_port = PORT_DEFAULT;

        /* wait to receive request from client */
       msgcode = msg_receive(&pInMsg->head,MSG_OPTION_NONE,0);
       if (msgcode != RCV_SUCCESS)
           printf("error %s in Receive, message will be ignored.\n",
                mach_errormsg((kern_return_t)msgcode));
       else
       {   if (pInMsg->head.msg_type == MSG_TYPE_EMERGENCY)
           {
              if (pInMsg->head.msg_id == NOTIFY_PORT_DELETED)
                  {  /* probably the death of a client’s reply */}
              else
                printf("Unexpected emergency message received: id is %d\n",
                        pInMsg->head.msg_id);
```

```
            }
           else  /* normal message */
           {
                 /* call server interface module */
               ok = random_server((msg_header_t *)pInMsg,
                                    (msg_header_t *)pRepMsg);
               if (pRepMsg->return_code != MIG_NO_REPLY)
               {
                   /* sending reply message to client */
                 pRepMsg->head.msg_local_port = pInMsg->head.msg_local_port;
                 pRepMsg->head.msg_remote_port = pInMsg->head.msg_remote_port;
                 msgcode = msg_send(&pRepMsg->head,MSG_OPTION_NONE,0);
                 if ((msgcode != SEND_SUCCESS) &&
                     (msgcode != SEND_INVALID_PORT))
                                /* Probably remote process death */
                     printf("error %s at Send.\n",
                                 mach_errormsg((kern_return_t)msgcode));
               }
           } /* normal message */
        } /* of message handling */
      }  /* of main loop */

}


main()
{
   port_t       ServerPort;
   kern_return_t retcode;

        /* add notification port to default port set */
   retcode = port_unrestrict(task_self_,task_notify_);
        /* allocate a service port */
   retcode = port_allocate(task_self(), &ServerPort);
   if (retcode == KERN_SUCCESS)
   {    /* add service port to default port set */
      (void) port_unrestrict(task_self(),ServerPort);
        /* check it in so users can find it */
      retcode  = netname_check_in(NameServerPort,"RandomServerPort",
                 PORT_NULL,ServerPort);
   }
   if (retcode != KERN_SUCCESS)
        printf("netname_check_in of RandomServerPort failed with code %s\n",
                        mach_errormsg(retcode));
   random_run ();
   printf("(* !!!!! Random server exited - give it up !!!!! *)\n");
   exit(2);
}
```

Table of Contents