Direction

ε

in

out

inout

requestport

replyport

sreplyport

ureplyport

waittime

msgoption

msgkind

ArgumentType

: *identifier*

: **NamedTypeSpec**

IntExp – **IntExp**

IntExp * **IntExp**

IntExp / **IntExp**

number

(**IntExp**)

RoutineDecl

Routine

SimpleRoutine

CamelotRoutine

Routine

routine *identifier* **Arguments**

SimpleRoutine

simpleroutine *identifier* **Arguments**

CamelotRoutine

camelotroutine *identifier* **Arguments**

Arguments

()

(A**rgumentList**)

ArgumentList

Argument

Argument; ArgumentList

Argument

Direction *identifier* **ArgumentType IPCFlags**

MACH_MSG_TYPE_MOVE_SEND

MACH_MSG_TYPE_MAKE_SEND_ONCE

MACH_MSG_TYPE_MOVE_SEND_ONCE

MACH_MSG_TYPE_PORT_RECEIVE

MACH_MSG_TYPE_PORT_SEND

MACH_MSG_TYPE_PORT_SEND_ONCE

IPCFlags

ε

IPCFlags, islong

IPCFlags, isnotlong

IPCFlags, dealloc

IPCFlags, notdealloc

PrevTypeSpec

identifier

VarArrayHead

array [] of

array [*] of

array [*: **IntExp**] of

ArrayHead

array [**IntExp**] of

StructHead

struct [**IntExp**] of

IntExp

IntExp + **IntExp**

BasicTypeSpec

IPCType

(**IPCType, IntExp IPCFlags**)

IPCType

PrimIPCType

PrimIPCType | PrimIPCType

PrimIPCType

number

polymorphic

MACH_MSG_TYPE_UNSTRUCTURED

MACH_MSG_TYPE_BIT

MACH_MSG_TYPE_BOOLEAN

MACH_MSG_TYPE_INTEGER_8

MACH_MSG_TYPE_INTEGER_16

MACH_MSG_TYPE_INTEGER_32

MACH_MSG_TYPE_CHAR

MACH_MSG_TYPE_BYTE

MACH_MSG_TYPE_REAL

MACH_MSG_TYPE_STRING

MACH_MSG_TYPE_STRING_C

MACH_MSG_TYPE_PORT_NAME

MACH_MSG_TYPE_POLYMORPHIC

MACH_MSG_TYPE_MOVE_RECEIVE

MACH_MSG_TYPE_COPY_SEND

MACH_MSG_TYPE_MAKE_SEND

uimport

simport

RCSDecl

rcsid *quotedstring*

TypeDecl

type **NamedTypeSpec**

NamedTypeSpec

identifier = **TransTypeSpec**

TransTypeSpec

TypeSpec

TransTypeSpec intran: *identifier identifier* (i*dentifier)*

TransTypeSpec outtran: *identifier identifier* (i*dentifier)*

TransTypeSpec destructor: *identifier (identifier)*

TransTypeSpec ctype: *identifier*

TransTypeSpec cusertype: *identifier*

TransTypeSpec cservertype: *identifier*

TypeSpec

BasicTypeSpec

PrevTypeSpec

VarArrayHead TypeSpec

ArrayHead TypeSpec

^ TypeSpec

StructHead TypeSpec

SubsystemMod

camelot

kerneluser

kernelserver

SubsystemName

identifier

SubsystemBase

number

WaitTime

waittime *string*

nowaittime

MsgOption

msgoption *string*

MsgKind

msgkind *string*

ServerPrefix

serverprefix *identifier*

UserPrefix

userprefix *identifier*

Import

ImportIndicant *filename*

ImportIndicant

import

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

Statements

ε

Statements Statement

Statement

Subsystem;

WaitTime;

MsgOption;

MsgKind;

ServerPrefix;

UserPrefix;

Import;

RCSDecl;

TypeDecl;

RoutineDecl;

skip;

;

Subsystem

subsystem **SubsystemMods SubsystemName SubsystemBase**

SubsystemMods

ε

SubsystemMods SubsystemMod

APPENDIX A

MIG Language Specification

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

Meta language description: The right hand side of a production rule is indented from its left hand side (its heading). Terminal symbols are in plain font. Non-terminal symbols are in bold font. Symbols in italics are non-terminal symbols not defined within the grammar but that have obvious definitions:

- *number* - a string of digits from 0 to 9
- *identifier* - a name consisting of upper and lower case letters, digits 0 to 9 and underscores, with the first character being a letter or an underscore
- *string* - a string of letters, numbers, slashes, periods, underscores, dollar signs and dashes
- *quotedstring* - a string of arbitrary characters (other than ‘’ and new-line) surrounded by quotes
- *anglestring* - a string of arbitrary characters (other than> and new-line) surrounded by angle brackets (<>)
- *filename* - a *quotedstring* or an *anglestring*

The symbolε indicates an empty string.

Symbols within a single production are separated by blanks. Alternative productions are listed on consecutive lines under a production rule heading.

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

```
        srandom (time ((int *) 0)); /* initialize random number generator */
        master ((int) random()% 16); /* create up to 15 slaves */
        return 0;
}
```

```
done.
*/

void
slave (int n)
{
        int i;

        for (i = 0; i < n; i += 1)
                cthread_yield();
        mutex_lock(lock);
        count –= 1;
        printf (“Slave finished %d cycles.\n”, n);
        condition_signal (done);
        mutex_unlock (lock);
}

/*
The master spawns a given number of slaves and then waits for them all to
finish.
*/

void
master (int nslaves)
{
         int i;

        for (i = 1; i <= nslaves; i += 1) {
                mutex_lock(lock);

/* Fork a slave and detach it, since the master never joins it individually. */

                count += 1;
                cthread_detach(cthread_fork(slave, random()% 1000));
                mutex_unlock (lock);
        }
        mutex_lock(lock);
        while (count!= 0)
                condition_wait (done, lock);
        mutex_unlock (lock);
        printf (“All %d slaves have finished.\n”, nslaves);
        cthread_exit(0);
}

int
main()
{
        count = 0;
        lock = mutex_alloc();
        done = condition_alloc();
```

```
buffer_t
buffer_alloc (int size)
{
        buffer_t b;
        extern char *malloc();

        b = (buffer_t) malloc (sizeof (struct buffer));
        mutex_init (&b–>lock);
        b–>size = size;
        b–>chars = malloc ((unsigned) size);
        b–>px = b–>cx = b–>count = 0;
        condition_init (&b–>non_empty);
        condition_init (&b–>non_full);
        return b;
}

#define BUFFER_SIZE 10

int
main()
{
         buffer_t b = buffer_alloc(BUFFER_SIZE);

        cthread_detach(cthread_fork(producer, b));
        cthread_detach(cthread_fork(consumer, b));
        return 0;
}
```

Single Master / Multiple Slaves

The following example shows how to structure a program in which a single master thread spawns a number of concurrent slaves and then waits until they all finish.

```
/*
Master/slave program structure.
*/

#include <stdio.h>
#include <cthreads.h>

int count; /* number of slaves active */
mutex_t lock; /* mutual exclusion for count */
condition_t done; /* signalled each time a slave finishes */

/*
Each slave just counts up to its argument, yielding the processor on each itera-
tion. When it is finished, it decrements the global count and signals that it is
```

```
void
buffer_put (char ch, buffer_t b)
{
        mutex_lock (&b–>lock);
        while (b–>count == b–>size)
                condition_wait (&b–>non_full, &b–>lock);
        ASSERT(0 <= b–>count && b–>count < b–>size);
        b–>chars [b–>px] = ch;
        b–>px = (b–>px + 1)% b–>size;
        b–>count += 1;
        condition_signal (&b–>non_empty);
        mutex_unlock (&b–>lock);
}

char
buffer_get (buffer_t b)
{
        char ch;

        mutex_lock (&b–>lock);
        while (b–>count == 0)
                condition_wait (&b–>non_empty, &b–>lock);
        ASSERT(0 < b–>count && b–>count <= b–>size);
        ch = b–>chars [b–>cx];
        b–>cx = (b–>cx + 1)% b–>size;
        b–>count –= 1;
        condition_signal (&b–>non_full);
        mutex_unlock (&b–>lock);
        return ch;
}

void
producer (buffer_t b)
{
        int ch;

        do buffer_put ((ch = getchar()), b);
        while (ch!= EOF);
}

void
consumer (buffer_t b)
{
        int ch;

        while ((ch = buffer_get (b))!= EOF)
                printf (“%c”, ch);
}
```

Associating Names with C Thread Objects

A name (actually, a pointer to a character string) can be associated with a C thread, a mutex or a condition variable. These names are not currently used for anything, but they can be of use to debugging. Such names are set via **cthread_set_name**, **mutex_set_name** and **condition_set_name**, and retrieved via **cthread_name**, **mutex_name** and **condition_name**.

Pitfalls of Preemptively Scheduled Threads

The C run-time library needs a substantial amount of modification in order to be used with preemptively scheduled threads. Currently the user must ensure that calls to the standard I/O library are serialized, through the use of one or more mutex variables. (The storage allocation functions **malloc** and **free** do not require any special precautions.)

The debuggers currently available under MACH cannot be used on programs that run with preemptively scheduled threads. Furthermore, the very act of turning on tracing or adding print statements may perturb programs that incorrectly depend on thread execution speed. One technique that is useful in such cases is to vary the granularity of locking and synchronization used in the program, making sure that the program works with coarse-grained synchronization before refining it further.

Examples

The following examples illustrate the use of the C threads facilities.

Hoare Monitors

This program demonstrates the use of these facilities to program Hoare monitors.

```
/*
Producer/consumer with bounded buffer.

The producer reads characters from stdin and puts them into the buffer. The
consumer gets characters from the buffer and writes them to stdout. The two
threads execute concurrently except when synchronized by the buffer.
*/

#include <stdio.h>
#include <cthreads.h>

typedef struct buffer {
        struct mutex lock;
        char *chars; /* chars[0...size–1] */
        int size;
        int px, cx; /* producer and consumer indices */
        int count; /* number of un-consumed chars in buffer */
        struct condition non_empty, non_full;
} *buffer_t;
```

relative execution speed of threads, the simplest solution is to share pointers to global or heap-allocated objects only. If a pointer to a local variable is shared, the procedure in which that variable is defined must remain active until it can be guaranteed that the pointer will no longer be de-referenced by other threads. The synchronization functions can be used to ensure this.

The programmer must remember that unless a library, like the standard C library, has been designed to work in the presence of re-entrancy, the operations provided by the library must be presumed to make unprotected use of shared data. Hence, the programmer must protect against this through the use of a mutex that is locked before every library call (or sequence of library calls) and unlocked afterwards.

Dynamic allocation and freeing of user-defined data structures is typically accomplished with the standard C functions **malloc** and **free**. The C threads package provides versions of these functions that work correctly in the presence of multiple threads.

Using the C Threads Package

All of the functions described have been implemented for the MACH multiprocessor operating system. Three implementations of threads are provided, in the form of libraries. Programs need not be re-compiled to use a different thread implementation, only re-linked. To compile a program that uses C threads, the user must include the file **cthreads.h**. (The directory **/usr/mach/include** should be in the user's **CPATH** search list for the C preprocessor to find this header file.) To link a program that uses C threads, the user must specify on the **cc** command line the **-lthreads** library, followed by the **-lmach** library. (The directory **/usr/mach/lib** should be in the user's **LPATH** search list for the linker to find these libraries.) Linking a program in this way is sufficient to initialize and begin use of the C threads package.

Debugging

Debugging with multiple C threads can be a problem for a variety of reasons:

- The order of thread context switching is not repeatable in successive executions of the program, so obvious synchronization bugs may be difficult to find.
- Since the program consists of multiple execution points, existing debuggers may not be usable.[1]
- The user may need to worry about concurrent calls to C library routines.

One way to avoid these problems is to set the number of kernel threads to 1 (**cthread_set_kernel_limit**). In this way, the various C threads act as co-routines and may be debugged more as a sequential program.

---

[1] A C thread version of **gdb** is in the works.

thread blocked in **mutex_lock** or **condition_wait** should be viewed as referencing the object continually, so freeing the object “out from under” such a thread is erroneous, and can result in bugs that are extremely difficult to track down.

A programmer can initialize an object of the struct *mutex* or struct *condition* referent type with **mutex_init** or **condition_init**, so that its address can be used wherever an object of type *mutex_t* or *condition_t* is expected. For example, the **mutex_alloc** function could be written in terms of **mutex_init** as follows:

```
mutex_t
mutex_alloc()
{
        register mutex_t m;
        m = (mutex_t) malloc (sizeof (struct mutex));
        mutex_init(m);
        return m;
}
```

Initialization of the referent type is most often used when the programmer has included the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation. For instance, a data structure might contain a component of type struct *mutex* to allow each instance of that structure to be locked independently. During initialization of the instance, the programmer would call **mutex_init** on the struct *mutex* component. The alternative of using a *mutex_t* component and initializing it using **mutex_alloc** would be less efficient.

mutex_clear and **condition_clear** allow the programmer to finalize an object of the struct *mutex* or struct *condition* referent type. For example, the **mutex_free** procedure could be written in terms of **mutex_clear** as follows:

```
void
mutex_free(m)
mutex_t m;
{
        mutex_clear(m);
        free ((char *) m);
}
```

Shared Variables

All global and static variables are shared among all threads: if one thread modifies such a variable, all other threads will observe the new value. In addition, a variable reachable from a pointer is shared among all threads that can de-reference that pointer. This includes objects pointed to by shared variables of pointer type, as well as arguments passed by reference in **cthread_fork**.

When pointers are shared, some care is required to avoid dangling reference problems. The programmer must ensure that the lifetime of the object pointed to is long enough to allow the other threads to de-reference the pointer. Since there is no bound on the condition will be true when the thread resumes, use of this procedure should always be of the form:

```
mutex_lock(m);
while (/* condition is not true */)
        condition_wait (c, m);
mutex_unlock (m);
```

Shared variables should be inspected on each iteration to determine whether the condition is true.

Precautions

One must take special care with data structures that are dynamically allocated and deallocated. In particular, if the mutex that is controlling access to a dynamically allocated record is part of the record, one must be sure that no thread is waiting for the mutex before freeing the record.

Attempting to lock a mutex that one already holds is another common error. The offending thread will block waiting for itself. This can happen when a thread is traversing a complicated data structure, locking as it goes, and reaches the same data by different paths. Another instance of this is when a thread is locking elements in an array, say to swap them, and it doesn't check for the special case that the elements are the same.

In general, one must be very careful to avoid deadlock. Deadlock is defined as the condition in which one or more threads are permanently blocked waiting for each other. The above scenarios are a special case of deadlock. The easiest way to avoid deadlock with mutexes is to impose a total ordering on the mutexes, and then ensure that threads only lock mutexes in increasing order.

One important issue the programmer must decide is what kind of granularity to use in protecting shared data with mutexes. The two extremes are to have one mutex protecting all shared memory, and to have one mutex for every byte of shared memory. Finer granularity normally increases the possible parallelism, because less data is locked at any one time. However, it also increases the overhead lost to locking and unlocking mutexes and increases the possibility of deadlock.

Management of Synchronization Variables

A mutex or condition variable can be allocated dynamically from the heap, or the programmer can take an object of the referent type, initialize it appropriately, and then use its address.

The functions **mutex_alloc** and **condition_alloc** provide dynamic allocation of mutex and condition objects.

mutex_free and **condition_free** allow the programmer to de-allocate mutex and condition objects that were allocated dynamically. Before de-allocating such an object, the programmer must guarantee that no other thread will reference it. In particular, a threads fetch the current value into a (thread-local) register, increment, and write the value back in some order, the counter will only be incremented once, losing one thread's operation. A mutex solves this problem by making the fetch-increment-deposit action atomic. Before fetching a counter, a thread locks the associated mutex. After depositing a new value, the thread unlocks the mutex. If any other thread tries to use the counter in the meantime, it will block when it tries to lock the mutex. If more than one thread tries to lock the mutex at the same time, the C threads package guarantees that only one will succeed; the rest will block.

While a mutex is locked, all other attempts to lock the mutex will block until the mutex is unlocked. When the mutex is unlocked, one of the blocked threads will then succeed in locking the mutex. By default, unlocking a mutex does not guarantee that one of the blocked threads will run immediately; normally the unlocking thread will continue to run and other threads will have to wait for assignment to kernel threads. A special mutex, however (made special by **mutex_set_special** and made normal by **mutex_clear_special**), has the property that a blocked thread will be made to run immediately (governed by kernel scheduling, of course).

A mutex variable is locked with the **mutex_lock** function. This function will block until the mutex can be (and is) locked. It is fairly common that a thread wishes to lock a mutex, but has other things it can do until it can succeed in locking the mutex. For these purposes, there is the **mutex_try_lock** function. If this function cannot lock the mutex, it simply returns instead of blocking (returning a status indicating this occurrence). The thread can proceed to accomplish other work and try its lock attempt later. A mutex is unlocked with **mutex_unlock**.

Condition Variables

Condition variables are used when one thread wants to wait until another thread has finished doing something. Every condition variable should be protected by a mutex. Conceptually, the condition is a boolean function of the shared data that the mutex protects. Commonly, a thread locks the mutex and inspects the shared data. If it doesn't like what it finds, it waits using a condition variable. This operation also temporarily unlocks the mutex, to give other threads a chance to get in and modify the shared data. Eventually, one of them should signal the condition (which wakes up the blocked thread) before it unlocks the mutex. At that point, the original thread will regain its lock and can look at the shared data to see if things have improved. It can't assume that it will like what it sees, because some other thread may have slipped in and mucked with the data after the condition was signaled.

The **condition_signal** function should be called when one thread wishes to indicate that the condition represented by the condition variable is now true. If any other threads are waiting (via **condition_wait**), then at least one of them will be awakened. If no threads are waiting, then nothing happens. The **condition_broadcast** function is similar to **condition_signal**, except that it awakens *all* threads waiting for the condition, not just one of them.

The **condition_wait** function takes as arguments a mutex *m* and a condition variable *c*. It unlocks *m*, suspends the calling thread until the specified condition is *likely* to be true, and locks *m* again when the thread resumes. Since there is no guarantee that the

A MACH thread is assigned to a *cproc* when one is free. By default, MACH threads are created so that each *cproc* has its own. However, the number of MACH threads can be limited (with **cthread_set_kernel_limit**). Making such a limit has a variety of effects:

- The maximum degree of true parallelism is set.
- The kernel overhead associated with the number of MACH threads is controlled.
- Allowing multiple C threads the chance to share a MACH thread reduces the overhead of synchronization between them by allowing context switches without kernel involvement.

It is possible to force a binding between a C thread and a MACH thread with **cthread_wire** (**cthread_unwire** undoes this). Wiring a C thread to a MACH thread keeps that C thread from being serviced by any other MACH thread; it also prevents that MACH thread from servicing any other C thread. In this way, a MACH thread will always be available to service this C thread, in the case where the number of MACH threads is being limited to less than the number of C threads.

The **cthread_yield** procedure is a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor. This may be required to avoid starvation when the number of MACH threads is limited.

When a C thread executes a blocking kernel function (**mach_msg** perhaps being the most extreme example), its corresponding MACH thread is blocked as well. To avoid blocking an excessive number of MACH threads, a special version of **mach_msg**, **cthread_mach_msg**, is provided. This function performs just like **mach_msg**, but it limits the number of C threads that can be "active" listeners for a port. A C thread is declared an active listener for a port when it calls **cthread_msg_active** or **cthread_mach_msg** specifying a port. It ceases to be an active listener when it calls **cthread_msg_busy** or **cthread_mach_msg** for a different port. If a C thread is not an active listener for a port when it calls **cthread_mach_msg** (intending to perform a message receive), and making it a listener would exceed the limit set for that port, the C thread itself will block (without blocking its underlying MACH thread), thereby providing more free (un-blocked) MACH threads. The C thread will get to execute its message receive operation when some other C thread stops being a listener for that port.

Synchronization

This section describes mutual exclusion and synchronization primitives, called mutexes and condition variables. In general, these primitives are used to constrain the possible interleaving of threads' execution streams. They separate the two most common uses of Dijkstra's *P* and *V* operations into distinct facilities. This approach basically implements monitors, but without the syntactic sugar.

Mutex Variables

Mutually exclusive access to mutable data is necessary to prevent corruption of data. As simple example, consider concurrent attempts to update a simple counter. If two

Thread Management

The sole means of creating new threads is via the **cthread_fork** function. This function returns a value of type *cthread_t*. The **cthread_self** function returns the *cthread_t* value that was returned by **cthread_fork** to the creator of the thread. This value forms a thread identifier that uniquely identifies the thread, and hence may be used as a key in data structures that associate user data with individual threads. Since thread identifiers may be reused by the underlying implementation, the programmer should be careful to clean up such associations when threads exit.

Although a thread automatically terminates when it returns from its top-level function, it may also terminate itself explicitly with **cthread_exit**.

The **cthread_detach** function indicates that the named thread will not rendezvous with any other thread. **cthread_join** is used to rendezvous with a thread's termination. When a rendezvous occurs, the joining thread proceeds, given the return value from the joined thread's top-level function, or the value supplied to **cthread_exit**, if that is the means of termination for the joined thread.

It follows that attempting to join one's own thread will result in deadlock.

The C threads package allows for a single value (of type *any_t*) to be associated with each thread, providing a simple form thread-specific "global" variable. This value is set by **cthread_set_data** and retrieved with **cthread_data**. More elaborate mechanisms, such as per-thread property lists or hash tables, can then be built with these primitives.

The current number of C threads can be obtained from **cthread_count**.

Execution of Threads

A set of C threads may execute in parallel on multiple processors within a system. Various factors, even beyond those involve kernel scheduling, affect the extent of parallelism achieved by a set of C threads.

Underlying C threads is the notion of *cproc*'s, and under *cproc*'s fall MACH threads. It is MACH threads that are actually managed by the kernel and therefore schedulable on processors. A C thread executes only when:

- A *cproc* has been assigned to the C thread
- A MACH thread has been assigned to the *cproc*.
- The kernel has decided to schedule the MACH thread.

(The issues involving the scheduling of MACH threads involves processor sets and task and thread assignments, policies and priorities and is not covered here.)

A *cproc* is assigned to a C thread when one is free. By default, *cproc*'s are created so that each C thread has its own. However, the number of *cproc*'s can be limited (with **cthread_set_limit**) so that the number of C threads that can be making progress is limited. A *cproc* becomes free when the C thread assigned to it terminates.

is used instead.)

Continuing the example of the *mutex_t* type, the functions **mutex_alloc** and **mutex_free** provide dynamic storage allocation and de-allocation. The functions **mutex_init** and **mutex_clear** provide initialization and finalization of the referent type. These functions are useful if the programmer wishes to include the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation. They should not be called on objects that are dynamically allocated via **mutex_alloc**. Type-specific functions such as **mutex_lock** and **mutex_unlock** are also defined, of course.

Initializing the C Threads Package

Initialization of the C threads package is fairly automatic, given the inclusion of **libthreads.a** when linking.

The header file **cthreads.h** defines the C threads interface. All programs using C threads must include this file.

The **cthread_init** function initializes the C threads implementation. This call is automatically made by **_start** if the C threads package was linked with the task.

Threads of Control

The C threads package allows for multiple threads of control in a C application.

Creation

When a C program starts, it contains a single thread of control, the one executing **main**. The thread of control is an active entity, moving from statement to statement, calling and returning from procedures. New threads are created by *fork* operations.

Forking a new thread of control is similar to calling a procedure, except that the caller does not wait for the procedure to return. Instead, the caller continues to execute in parallel with the execution of the procedure in the newly forked thread. At some later time, the caller may rendezvous with the thread and retrieve its result (if any) by means of a *join* operation, or the caller may *detach* the newly created thread to assert that no thread will ever be interested in joining it.

Termination

A thread *t* terminates when it returns from the top-level procedure it was executing.[1] If *t* has not been detached, it remains in limbo until another thread either joins it or detaches it; if *t* has been detached, no rendezvous is necessary.

---

[1] This is also effectively true of the initial thread (executing **main**). After effectively terminating itself, the initial thread waits for all other threads to terminate. It then terminates the task.

CHAPTER 3 C Threads

This chapter describes the C threads package.

Introduction

MACH provides a set of low-level, language-independent primitives for manipulating threads of control. The C Threads package is a run-time library that provides a C language interface to these facilities. The constructs provided are similar to those found in Mesa and Modula-2+, forking and joining of threads, protection of critical regions with mutex variables and synchronization by means of condition variables.

Naming Conventions

An attempt has been made to use a consistent style of naming for the abstractions implemented by the C Threads package. All types, macros, and functions implementing a given abstract data type are prefixed with the type name and an underscore. The name of the type itself is suffixed with *_t* and is defined via a C typedef. Documentation of the form:

typedef struct *mutex* {...} **mutex_t*;

indicates that the *mutex_t* type is defined as a pointer to a *referent type* struct *mutex* which may itself be useful to the programmer. (In cases where the referent type should be considered *opaque*, documentation such as:

typedef... *cthread_t*;

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 **mach_msg** with both the MACH_SEND_MSG and MACH_-RCV_MSG options; *R* use a pair of calls, **mach_msg** with the MACH_SEND_MSG option and **mach_msg** with the MACH_-RCV_MSG option. 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 in the *sys*Server.c code. The layout of a symbol table (*mig_symtab_t*) is defined in **<mig_errors.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 server call interface that is re-directed at run-time ala **syscall** (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 **operation-name.c**.
–server *name*	name the server file *name*.
–user *name*	name the user file *name*.
–header *name*	name the header file *name*.
–iheader *name*	name the user internal header file *name*. This is only used for *kernelserver* subsystems.

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 *sys***.d** file to show the dependencies of the *sys***.h**, *sys***User.c** and *sys***Server.c** on the *sys***.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 *sys***.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**, *random***User.c** and *random***Server.c** as output. If the “–MD” switch was given, a *random***.d** file will also be generated.

when a task is created and can be found in various globals. One of these ports is the port to the Net Name Server which can be used to check-in or look-up user supplied ports. The *replyport* for MIG is a per-thread global that is initialized when the thread is created. The *replyport* can be specified as a parameter to an operation if necessary.

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 de-allocate 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 several things:

- The *requestport* parameter is an invalid port or lacks send rights.
- 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.
- 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. This set of functions is provided by the **libmach.a** function **mach_msg_server**. 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 MACH_PORT_NULL).

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>
```

subsystem *subsystemmod sys message-base-id;*

KernelServer Subsystem

The *KernelServer* environment is that in which a routine which is to be the target server routine associated with a MIG definition is to reside in the MACH kernel. The target server routines have the same prototypes as they would otherwise.

A *KernelServer* subsystem modifies the behavior of MIG type translation in that the server stub expects the types to have been translated on the way to it, but it sets the *msgt_name* field on return to the IPC type associated with *receiving-IPCType*, instead of expecting the type to be changed magically.

Various name conflicts can arise when the client and server code exist in the same space. One method to resolve this conflict available for *KernelServer* subsystems is to redefine the names of the client stub routines. This is done with an internal header file. This file consists of one line for each operation *foo*, of the form:

#define *foo foo*_external

This file can be **#included** into a file using the client interface, to force it to call the MIG client stubs instead of the target server routines, if they would otherwise have the same name. When an internal header file is generated (via the *-iheader* MIG option), this file is automatically **#include**d in the user stub file.

KernelUser Subsystem

This is the environment in which a routine in the MACH kernel wishes to call what would be the client stub. The client stub for this environment has the same prototype as it would otherwise. It differs only in that it uses a MACH kernel routine to perform the MACH IPC operation; an internal kernel port is used as the reply port.

Camelot Subsystem

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.

See the Camelot documentation Guide to the Camelot Distributed Transaction Facility for a full description of how *camelotroutine*s differ from *routine*s.

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 on which to call the server. The service ports for basic system servers are inherited field This parameter is of type **mach_msg_type_number_t** for *in* parameters, and ***mach_msg_type_number_t** for *out* or *inout* parameters.

The C type used for the data parameter is chosen as follows:

- A *type-name* that is a built-in *IPCType,* where *IPCType* is other than MACH_MSG_TYPE_STRING or MACH_MSG_TYPE_STRING_C, has *type-name* as its C data parameter type for *in* parameters, and **type-name* for *out* or *inout* parameters.
- A *type-name* that is a built-in *IPCType* of MACH_MSG_TYPE_STRING or MACH_MSG_TYPE_STRING_C has *type-name* as its C data parameter type, regardless of the direction of the parameter.
- A *type-name* that is an array type has *type-name* as its C data parameter type, regardless of the direction of the parameter.
- A *type-name* that is a structure type has *type-name* as its C data parameter type for *in* parameters, and **type-name* for *out* or *inout* parameters.
- A *type-name* that is a pointer type has *type-name* as its C data parameter type for *in* parameters, and **type-name* for *out* or *inout* parameters.

A parameter (set) will be generated for each parameter listed in the operation definition, with the exception of *waittime*, *msgkind* and *msgoption* specifications.

The *requestport* parameter is always listed explicitly as a parameter. The *requestport* specification merely allows the specification of which parameter in the list is the request port; the default is the first parameter.

The presence of the *replyport* parameter causes an explicit parameter to appear in the argument list for the client stub routine, as well as the corresponding argument to appear in the call to the target server routine, to pass an explicitly named reply port. If *replyport* is not specified, a reply port private to the MIG interface routines will be used; it will not appear as a parameter in any calls.

The *replyport* parameter causes a reply port to be named in both the client stub routine, and the target server routine. The *sreplyport* and *ureplyport* parameters can be used if only one of these parameters is needed. When a *ureplyport* parameter is present, the client stub routine will have an extra parameter through which it can name an explicit reply port. However, the target server routine will not have this extra parameter; the reply port will be known only to the MIG generated server function. When a *sreplyport* parameter is named, the client stub routine will not have an extra parameter through which an explicit reply port is named. Instead, the MIG local reply port will be used. However, the target server routine will be passed this reply port as an explicit argument (instead of its being known only inside the MIG server stub).

Special Environment Subsystems

MIG generates alternate code for the client or server stubs for various special environments. MIG generates this code when a subsystem modifier is specified in the subsystem specification:

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. This value sets the *timeout* argument to the **mach_msg** call.

MsgKind Specification

The *msgkind* specification is of the form:

msgkind *manifest-constant;*

where the *manifest-constant* may be one of the values from the file **<message.h>**. The currently available types are MACH_MSGH_KIND_NORMAL (the default) and MACH_MSGH_KIND_NOTIFICATION. The value MACH_MSGH_KIND_NORMAL can be used to reset the *msgkind* option. The value of this specification is used to set the *msgh_kind* field in the IPC header.

MsgOption Specification

The *msgoption* specification is of the form:

msgoption *manifest-constant;*

where the *manifest-constant* may be one of the option values from the file **<message.h>**. This value is OR'ed into the *option* argument to the **mach_msg** call. The value MACH_MSG_OPTION_NONE can be used to reset the *msgoption* option.

Function Prototype Generation

The prototype for a user stub or target server function is derived from the MIG operation definition. In a general way, the argument list for the prototype is the same as that for the MIG operation, with the MIG type names replaced by the *UserType* or *TransType* names, when defined.

A MIG parameter can expand into at most three actual parameters:

- The first parameter in the prototype is the data parameter derived from the MIG type, as explained below.
- If the MIG type is or derives in any way from MACH_MSG_TYPE_POLYMORPHIC, an extra parameter of type **mach_msg_type_name_t** is added after the data parameter for *in* parameters, and of type ***mach_msg_type_name_t** for *out* or *inout* parameters. This parameter carries the actual MACH IPC type of the data parameter.
- If the MIG type is or derives in any way from a variable sized array, an extra parameter is added after the data parameter (or after the polymorphic data type, if present) to contain the actual array size (the number of elements). MIG scales this array element count when calculating the *msgt_number* filed in the message type

Operation Descriptions

An operation description 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** [,**IPC-flags**]

where *specification* may either be omitted or be one of the following: *in* | *out* | *inout* | *requestport* | *replyport* | *sreplyport* | *ureplyport* | *waittime* | *msgkind* | *msgoption*. **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. The **IPC-flags** are either omitted or consist of a comma separated list of *dealloc*, *notdealloc*, *islong* and *isnotlong*. These flags will override any IPC flags associated with the type definition.

The *operation-type* is either *routine*, to specify an operation that expects a reply, or *simpleroutine*, for an operation that does not receive a reply.

Request and Reply Ports

The first unspecified parameter in any operation statement is assumed be the *requestport* unless a *requestport* parameter is specified. This is the port to which the message will be sent.

By default, MIG user stub routines use a MIG internal reply port for receiving the reply message. The *replyport* specification allows one of the parameters to be an explicitly named reply port. The *sreplyport* and *ureplyport* specifications have a related effect that is related to the generation of the actual function interface prototypes, as explained below.

Parameter Direction

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.

WaitTime Specification

The *waittime* specification has two forms:

waittime *time;*

nowaittime;

```
#define  MACH_MSG_TYPE_MOVE_RECEIVE         16
#define  MACH_MSG_TYPE_MOVE_SEND            17
#define  MACH_MSG_TYPE_MOVE_SEND_ONCE       18
#define  MACH_MSG_TYPE_COPY_SEND            19
#define  MACH_MSG_TYPE_MAKE_SEND            20
#define  MACH_MSG_TYPE_MAKE_SEND_ONCE       21
#define  MACH_MSG_TYPE_POLYMORPHIC          -1
```

the following built-in MIG types can be viewed as defined in the following way from the MACH IPC types:

```
type     MACH_MSG_TYPE_MOVE_RECEIVE =       16;
type     MACH_MSG_TYPE_MOVE_SEND =          17;
type     MACH_MSG_TYPE_MOVE_SEND_ONCE =     18;
type     MACH_MSG_TYPE_COPY_SEND =          19 | 17;
type     MACH_MSG_TYPE_MAKE_SEND =          20 | 17;
type     MACH_MSG_TYPE_MAKE_SEND_ONCE =     21 | 18;
```

The other major use of transmission type changes is to specify a parameter (most likely a port type) that is polymorphic on the sender's side (client side for *in* parameters, server side for *out* parameters) but of a known type on the receiver's side. That is, the sender will specify a port and a port type, but the result of sending this port will be to generate a known port type. The following built-in MIG types have such a specification:

```
type     MACH_MSG_TYPE_PORT_RECEIVE =       -1 | 16;
type     MACH_MSG_TYPE_PORT_SEND =          -1 | 17;
type     MACH_MSG_TYPE_PORT_SEND_ONCE =     -1 | 18;
```

As an example, **std_types.defs** makes the following definition:

```
type mach_port_send_t = MACH_MSG_TYPE_PORT_SEND
                        ctype: mach_port_t;
```

The (kernel) function **device_set_filter** is defined as follows:

```
routine device_set_filter(
                        device: device_t;
                        receive_port: mach_port_send_t;
                        priority: int;
                        filter: filter_array_t, IsLong
                        );
```

On the sending side, *receive_port* is of MIG type *polymorphic*, so it expands into an argument of type *mach_port_t* (the *ctype*) and an argument to specify the MIG type for the message header (*mach_msg_type_name_t*). The sender can supply a receive or send right, specifying MACH_MSG_TYPE_MAKE_SEND or MACH_MSG_TYPE_-COPY_SEND, respectively, for the port type parameter. On the receive side, *receive_-port* will automatically become of IPC type MACH_MSG_TYPE_MOVE_SEND, and the receiver will only see a single parameter, or C type *mach_port_t*.

In the example, **task_deallocate** is called on the translated input parameter, *task_t*, after the return from the server procedure and can be used to de-allocate any or all parts of the internal variable.

Destructor functions are called only for *in* parameters. They are not called if the parameter is also an *out* argument, because the correct time to de-allocate an *out* parameter is after the reply message has been sent, which is not code that can be generated by MIG. (In these circumstances, *dealloc* flags are needed.)

A *destructor* function can be used even if no translation is specified. For example, if a large out-of-line data segment is passed to the server it could use a *destructor* function to de-allocate the memory after the data was used.

Note that translation functions are called only for scalar values; MIG will not generate code to translate the multiple elements of an array.

Transmission Type Changes

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

> type *type-name* = *sending-IPCType* | *receiving-IPCType*;

This syntax indicates that the *msgt_name* field in the message should be set to the IPC type associated with *sending-IPCType* when the message is transmitted, and that, in some magic way, the *msgt_name* field will be the IPC type associated with *receiving-IPCType* when it is received. Such an IPC type change clearly only occurs when the associated parameter is handled by some intermediary, or when it is handled by the kernel.

In cases other than a *KernelServer* subsystem, when this type is sent (either as an *in* parameter or the return of an *out* or *inout* parameter), the *msgt_name* field is set by the sender of the message to the IPC type associated with *sending-IPCType* and the receiver expects the field to have the IPC type value associated with *receiving-IPCType*. This has the expected behavior for *in* and *out* parameters, namely, that the sender and receiver of the parameter sees the sending and receiving IPC types, respectively. For *inout* parameters, this is probably not what is desired, since the client will see the *inout* parameter return with a different IPC type then that with which it was sent.

Note that this syntax only expresses that the IPC type, as reflected in the *msgt_name* field, will be changed; it does not express an actual type change as seen by the client stub or target server routine, nor does it express how such a conversion occurs. If this feature is used with other than port types, it will probably be used in conjunction with *cusertype* and *cservertype* to express the probable (magic) type conversions that will be occurring.

Transmission type changes are useful in *KernelServer* subsystems where the recipient of a message is the kernel and so the kernel can arrange for IPC type translations. This is most likely to be used in conjunction with request and reply ports, which are translated by necessity in the transmission process. For example, given the following definitions for the MACH IPC port types:

Note that specifying an *intran* function implicitly sets both *ServerType* and *TransType*. A call to the *intran* function is inserted automatically by MIG before the call to the target server routine. The actual translation function must be “hand coded” and imported, with the following prototype:

TransType **intran-function** (*ServerType x*);

Translation back from the *TransType* to the *ServerType* is performed by an *outtran* function. The syntax is:

outtran: *ServerType* **outtran-function** (T*ransType)*

Note that specifying an *outtran* function implicitly sets both *ServerType* and *TransType*. A call to the *outtran* function is inserted automatically by MIG after the call to the target server routine. The actual translation function must be “hand coded” and imported, with the following prototype:

ServerType **outtran-function** (*TransType x*);

Consider an example for task related kernel primitives:

type *task_t* = *mach_port_t*
    intran: *task_t* **convert_port_to_task**(*mach_port_t*)
    outtran: *mach_port_t* **convert_task_to_port**(*task_t*)
    destructor: **task_deallocate** (*task_t*);

(The destructor function will be explained shortly.) In this example, *mach_port_t*, which is the type seen by the user code, is defined as a port in the message. The type seen by the target server code is *task_t*, which is some data structure used by the server (the kernel, in this case) to store information about each task it is serving. The *intran* function, **convert_port_to_task**, translates values of type *mach_port_t* to *task_t* on receipt by the server process. The *outtran* function, **convert_task_to_port**, translates values of type *task_t* to type *mach_port_t* before return.

For this example to work within the kernel, the kernel must hold some valid reference (and therefore, some lock) upon its *task_t* object. Imagine that the *intran* function will do this. This reference must be released after the target server routine is called. When *intran* functions are called, it is fairly common that the translation results in allocated storage, or perhaps additional reference counts for some objects that must be released. For this purpose, there are *destructor* functions. The syntax is:

destructor: **destructor-function** (T*ransType)*

Note that specifying a *destructor* function implicitly sets *TransType*. A call to the *destructor* function is inserted automatically by MIG after the call to the target server routine. The actual destructor function must be “hand coded” and imported, with the following prototype:

void **destructor-function** (*TransType x*);

type polymorphic = (MACH_MSG_TYPE_POLYMORPHIC,32);

Type translation information

Optional information describing procedures for translating the type or destroying values 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 de-allocate input or translated types after they have been used.

For each MIG type, there are up to three corresponding C types. These are:

- The type used in the client stub module (the *UserType*), which is the value that will define the type seen in the prototype to the user side stub module and the type used for inserting data into the outgoing IPC message to the server.
- The type seen by the server stub module (the *ServerType*) when it extracts this value from the incoming IPC message. It is also the type used by the server stub when inserting a value into an outgoing (return) IPC message. No conversion functions are called to convert between the *UserType* and the *ServerType*; these types must be effectively equivalent (at least in storage requirements) and are simply different names given by the different environments to the same storage.
- The translated type used by the server procedure (the *TransType*). This is the value used to define the type seen by the target routine called in the server.

By default these three types are all the same, and have the same name as the MIG type.

The *UserType* and *ServerType* may be specified with the *cusertype* and *cservertype* options, respectively, or both specified together with the *ctype* option. The *TransType* defaults to the *ServerType*; it can not be set explicitly; it is set implicitly by the specification of translation functions.

Examples:

type *mach_port_move_receive_t* = MACH_MSG_TYPE_MOVE_RECEIVE
    ctype: *mach_port_t*;

type *funny_int* = *int*
    cusertype: *user_int*
    cservertype: *server_int*;

If the *ServerType* and the *TransType* are to be different, then translation functions must be defined to perform the conversion. All data type translations (and the destructor function as well) are performed on the server side.

Translation from the *ServerType* to the *TransType* is performed by an *intran* function. The syntax is:

intran: *TransType* **intran-function** (S*erverType)* 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 (as if with **vm_allocate**). It is up to the user to de-allocate this memory when the user is finished with the data.

The C type name is assumed to resolve to a C type of form *... MIG will generate code to indirect through the parameters of these types to access the actual data.

Examples of Complex Types:

```
type procids = array [10] of int;

type procidinfo = struct [5*10] of MACH_MSG_TYPE_INTEGER_32;

type vardata = array [*: 1024] of int;

type array_by_value = struct [1] of array [20] of
                              MACH_MSG_TYPE_CHAR;

type page_ptr = ^ array [4096] of MACH_MSG_TYPE_INTEGER_32;

type var_array = ^ array [] of int;
```

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 run-time. 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 = (MACH_MSG_TYPE_POLYMORPHIC,32);

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

and then in client C code

```
mach_port_t server, port;
kern_return_t kr;

kr = SendPortOrInt (server, 5, MACH_MSG_TYPE_INTEGER_32);
kr = SendPortOrInt (server, port, MACH_MSG_TYPE_COPY_SEND);
```

The built-in type *polymorphic* is effectively defined as:

The MIG generated code assumes that the C types *my_string* and *kern_return_t* 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 MACH_MSG_TYPE_STRING or MACH_MSG_TYPE_STRING_C is declared as a C char **foo* or char *foo*[*n*]. Thus it generates code for a parameter passed by reference and uses **strncpy** for assignment statements.

Structured Types

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

> struct [*size*] of **comp-type-desc**
>
> array [*size*] of **comp-type-desc**
>
> array [*: *maxsize*] 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.

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. assuming 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. Other than the difference in argument passing, the fixed size *struct* and *array* declarations are the same. There is no way to specify the fields of a C structure to MIG. The *size* and **comp-type-desc** are just used to give the size of the structure.

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 in-line *inout* arguments are not supported. The **comp-type-desc** must be of fixed size (i.e., not a variable sized array).

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 which cases the array or structure is of variable size, and a parameter is defined immediately following the array parameter to 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* [, **IPC-flags**])

or:

ipc-typename

The *user-typename* is the name of the MIG 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 optional IPC option keywords which are: *dealloc, notdealloc*, *islong* and *isnotlong*. The user may define additional types. If the *ipc-typename* is one of the standard ones, except for MACH_MSG_TYPE_STRING, MACH_MSG_TYPE_STRING_C, MACH_MSG_-TYPE_UNSTRUCTURED or MACH_MSG_TYPE_REAL, just the *ipc-typename* can be used.

There are two sets of IPC flags: one set pertains to de-allocation of the parameter, and one pertains to normal versus long form type descriptors.

The de-allocation flag is used to describe the treatment of ports and pointers after the messages with which they are associated have been sent. *dealloc* causes the de-allocation 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 de-allocated after the message is sent. If *dealloc* is used with a pointer, the memory that the pointer references will be de-allocated after the message has been sent. An error will result if *dealloc* is used with any argument other than a port or a pointer. The use of *dealloc* with ports is not normally needed, as there are built-in types that dispose of ports properly.

Normally, MIG will choose the appropriate size for the type descriptor (**mach_msg_-type** or **mach_msg_type_long**). The *islong* keyword forces a long form type descriptor to be used for a parameter of this type. If a long form type descriptor was not needed, a warning is issued, but a long form type descriptor is used anyway. *isnotlong* specifies that a normal form type descriptor is to be used. If a long form type descriptor is needed, a warning is issued, and a long form type descriptor will be used.

Some examples of simple type declarations are:

type *int* = MACH_MSG_TYPE_INTEGER_32;

type *my_string* = (MACH_MSG_TYPE_STRING,8*80);

type *kern_return_t* = *int*;

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.

ServerPrefix Specification

The *serverprefix* specification is of the form:

> serverprefix *string;*

The word *serverprefix* is followed by an identifier *string* which will be prefixed 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. This specification applies to all operations defined after this specification in the definition file.

UserPrefix Specification

The *userprefix* specification is of the form:

> userprefix *string;*

The word *userprefix* is followed by an identifier *string* which will be prefixed 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. This specification applies to all operations defined after this specification in the definition file.

Options Declarations

Along with *serverprefix* and *userprefix*, 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. 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 defined options are:

- *waittime* specification — wait time to be used by the **mach_msg** call.
- *msgkind* specification — special kinds of messages
- *msgoption* specification — special options for the **mach_msg** call

Type Specifications

All types referenced in a definition file must be defined if they are other than the built-in IPC types.

Simple Types

A simple type declaration is of the form:

import “**my_defs.h**”;

import “**/usr/mach/include/cthreads.h**”;

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.

Standard Operations

There are two types of standard operations that may be specified:

- Routine
- Simple Routine

Also, 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 *msgh_id* number.

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**

where **specification** may either be omitted or one of the following:

- the direction of the parameter (*in*, *out* or *inout*)
- a declaration of the parameter as an explicit version of an otherwise implied value (*requestport, replyport, sreplyport, ureplyport*)
- a property of the generated **mach_msg** call (*waittime, msgkind, msgoption*)

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. The **type-description** may be appended with additional IPC flags to override any IPC attributes defined in the **user-typename**.

A *simpleroutine* sends a message to the server but does not expect a reply. The return value of a *simpleroutine* (type *kern_return_t*) is the value returned by the **mach_msg** primitive send operation. A *simpleroutine* is used when asynchronous communication with a server is desired. *simpleroutine*s cannot have *out* or *inout* parameters.

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

MACH IPC type (which is placed in the *msgt_name* field in a data descriptor) will be MACH_MSG_TYPE_INTEGER_32. MIG will set the *msgt_size* filed accordingly (in this case, 32 bits), and will compute the value for *msgt_number* on the basis of *msgt_size*, given the array and structure sizes given in the MIG type declaration.

- The data types used in the generated C code. The MIG type name is used, by default, whenever a C type name is used. The C types to be used can be directly specified. Either way, MIG separates the notions of its type name by which it refers to parameters and the type names used in the generated C code. Regardless of whether the MIG type name is used or a user specified type name is used for the C data types, the C data types must be specified in some header file "imported" into the generated C code, as well as the users of this code.

The **type-desc** defines the properties of the MIG type, such as its derivation from basic MIG types, and the storage attributes for the type. It also specifies attributes that will implicitly or explicitly set various IPC flags in generated type descriptors:

- *msgt_inline* — data follows the type descriptor (as opposed to a pointer to the data)
- *msgt_longform* — type descriptor is the long form (**mach_msg_type_long** as opposed to **mach_msg_type**)
- *msgt_deallocate* — de-allocate related port or storage

The optional **translation-info** defines the C types to use, as well as translation functions to be called to translate data types in the generated server stub.

MIG understands the MACH IPC types as built-in types. It allows the construction of types from these built-in types as structures of, arrays of and pointers to these basic or generated types.

Import Declarations

If any of the type names to be used in the generated C code 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** statements, e.g. **<***file-name***>** or **"***file-name***"**.

For example:

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 (*msgh_id* field) 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.

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 to the input string.

Type Declarations

A type declaration is of the form:

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

The purpose of type declarations is to inform MIG about the types to use for the parameters of the various operation routines, and to inform MIG of the storage requirements for parameters and the method by which the parameters are to be “marshalled” to form the data for an IPC message. It is possible to describe the types of the parameters for a routine directly in the declaration of a routine; but, it is often more convenient to define the types and their attributes separately, giving this collection of information a **user-typename**, and then just simply use this **user-typename** when specifying a routine parameter.

The *user-typename* is the name by which MIG will refer to the type. It must eventually be defined in terms of built-in MIG types, although **<mach/std_types.defs>** and **<mach/mach_types.defs>** already make most of the needed definitions. A MIG type definition really defines three related notions:

- The MIG type name itself, used in making new MIG type declarations.
- The underlying MACH IPC type. For example, if a MIG type is defined as an array of structures of the basic type MACH_MSG_TYPE_INTEGER_32, then the

fault, given a file defining interfaces with names *foo*, *bar*, etc. to subsystem *sys*, the output of MIG will be as follows:

- The user interface module will consist of C functions named *foo*, *bar*, etc. that gather the parameters defined for these operations and generate a MACH IPC message for the desired port.
- The user header module will consist of the necessary data descriptions, and the function prototypes for these C functions. The prototypes for these functions are derived from the MIG specification in a way described at length below.
- The server interface module will consist of a top-level C function called *sys***_server**, appropriate for use by **mach_msg_server**. This generated routine has the following prototype:

  boolean_t *sys***_server** (mach_msg_header_t **InHeadP*, mach_msg_header_t **OutHeadP*);

  (The actual parameters are of type **mig_reply_header_t** which are cast from the supplied **mach_msg_header_t**.) This routine examines the given message header to see if the message is one recognized by this subsystem (by examining *InHeadP–>msgh_id*). If the message is not recognized, FALSE is returned by *sys***_server**. If the message is recognized, the parameters from the IPC message are unpacked and the corresponding routine (i.e., *foo*, *bar*, etc.) will be called, with TRUE returned by *sys***_server**. The return code from the called function will be returned as the *RetCode* field in the output message. By default, the routines called in the server have the same names as those called by the client since it is assumed that the client and server live in separate spaces.

MIG provides many controls over this generation process, including changing the names of the various generated entities.

MIG Specification File Basics

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

- Subsystem identification
- RcsId specification
- Type specifications
- Import declarations
- Operation descriptions
- 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.

CHAPTER 2 MACH Interface Generator (MIG)

This chapter describes MIG, the MACH Interface Generator.

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 target server procedure or function returns, the generated interface module gathers the output parameters and correctly formats a reply message.

A given MIG specification file defines either some interfaces to a *subsystem*, or the data types of values to be passed to or returned from that subsystem, or both. By de-

CHAPTER 1 Introduction

This document describes information of generic interest to anyone who is writing a MACH server, or any task that wishes to communicate with another task. Whereas the Kernel Interface document describes the “pure kernel” which any task must use (possibly hidden under multiple emulation layers), this document describes facilities that a task does not *need* in an absolute sense, but probably wishes to use to work in the MACH environment.

This first part starts with a description of MIG (the MACH Interface Generator), which is used to specify the operations performed by a server, and the parameters required for the messages between clients and servers. This translator is described in this document for two reasons:

- Although the kernel uses MIG (the MACH Interface Generator) to generate the format of messages sent to it, one could simply view the kernel as taking messages of a format chosen arbitrarily. Since MIG is itself an application program (translator), not a kernel service, it is not documented in the Kernel Interface document.
- A server writer need not use MIG for anything. The format of kernel messages is fixed, so a server writer really needs MIG only if the writer wishes to use MIG for generated RPC message formats. Note that, of course, all supplied MACH 3 servers expect and return MIG formatted messages.

C threads is also covered in this document, since most servers will probably be written as multi-threaded applications.

Contents

This book is in the Open Software Foundation Mach 3 series.

Books in the OSF Mach 3 series:

MACH 3 Kernel Principles

MACH 3 Kernel Interface

Server Writer's Guide

Server Writer's Interface

Revision History:

Revision 0	November 3, 1990	Member's Meeting Version
Revision 0.1	May 6, 1991	Split Version

.

This document is partially derived from earlier Mach documents written by Richard P. Draves, Michael B. Jones, Mary R. Thompson, Eric Cooper, Richard P. Draves and Randall Dean.

Server Writer's Guide

Open Software Foundation and Carnegie Mellon University

Keith Loepere, Editor

May 6, 1991