OSF Research Institute

Advanced Development
One Cambridge Center - Cambridge, MA 02142 - USA

Untyped MIG: What Has Changed and Migration Guide

Franco Travostino
(travos@osf.org)

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

1. Introduction

The goal of this document is to highlight the differences between the CMU MACH3 MIG and the Untyped MIG ([Loepere92] and [Loepere93] contain the full description of both versions).

Untyped MIG implements the new MACH IPC interface defined in [Loepere93a]: since the new MACH IPC has been nick-named Untyped IPC (no more type tags to describe each argument), MIG has inherited the "Untyped" attribute.
In addition, Untyped MIG implements the new MIG protocol [Travos93]: it encodes data accordingly to the Network Data Representation (NDR) protocol [Kong91][DCE91], and it has a new set of end-to-end rules to marshal/unmarshal data.

Untyped MIG provides several new options:

- Options that were dictated by the new MACH IPC interface (e.g., Overwrite, Physicalcopy, ServerSecToken).
- Options that were implemented during the massive revamping of the MACH3 MIG: extensions to the language, performance optimizations, utilities (e.g., more complex types, SameCount, logging, etc.).

The first chapter documents these options.

In few cases, some options have been revoked, because they did not match any longer with the MACH IPC interface, or because they were related to obsolete MIG syntax: the second chapter goes through the list of keywords that have been pruned.

The third chapter, "Migration Guide", address the problem of converting the MIG Description Language scripts (.defs) from MACH3.0 MIG Language to Untyped MIG Language.

Untyped MIG is part of the NMK15/MK5.0 distributions (4/15/93). The NMK15 MACH kernel and the MK5.0 OSF/1 Server are also the first deliverables to be based on Untyped IPC and to be built with Untyped MIG.

2. Extensions

2.1 Name

The name of the command has been preserved ("mig"). In fact, mig corresponds to the shell script front end, that invokes CPP and the "migcom" binary afterwards. The name of the "migcom" binary has been changed: it is specified by the environment variable MIGCOM_UNTYPED (default: /usr/mach3/ccs/lib/migcom_untyped).

2.2 Command line Options

There are some additions to the set of command line options:

-k|-K Do or don't enforce the generation of ANSI C code only, without any #ifdef. With -K, the stubs have both the K&R syntax and the ANSI C syntax, through the option "#ifdef __STDC__" (default = -k)

-L|-l Do or don't insert the code for logging errors, and capturing traffic metrics (default = -l). Two examples of code generated with the -L option follows:

Example 1

```
#if     TypeCheck
        if (In0P->par2_in.type != MACH_MSG_OOL_DESCRIPTOR) {
#if MIG_DEBUG
                LOG_ERRORS(MACH_MSG_LOG_SERVER,
                        MACH_MSG_ERROR_WHILE_PARSING,
                        ("par2_in"), "MIG_GUIDE", 119)
#endif /* MIG_DEBUG */
                MIG_RETURN_ERROR(OutP, MIG_TYPE_ERROR);
        }
#endif  /* TypeCheck */
```

Example 2

```
#ifdef LOG
        LOG_TRACE(MACH_MSG_LOG_USER,
                MACH_MSG_REQUEST_BEING_SENT,
                InP->Head.msgh_id,
                InP->Head.msgh_size,
                InP->msgh_body.msgh_descriptor_count,
                0, /* RetCode */
                /* Ports */
```

```
                              0,
                              /* Out-of-Line Ports */
                              0,
                              /* Out-of-Line Bytes */
                              8192 + InP->par1_in[0].size + InP->par1_in[1].size +
                                   InP->par1_in[2].size + InP->par1_in[3].size +
                                   InP->par1_in[4].size,
                              "MIG_GUIDE", 145);
#endif /* LOG */
```

In Example 1, the code under MIG_DEBUG allows to detect which argument failed the type checking (the standard code would return only the anonymous MIG_TYPE_ERROR). In Example 2, the probe is able to account for the size of both fixed and variable sized arguments. It is left to the user to define the compile time variables MIG_DEBUG, LOG_ERRORS, LOG and LOG_TRACE (if the compile switches are not turned on, the binaries generated from stubs with or without probes are the same).

-dheader <file> In the past, it has been impossible to evict from the stubs a #define corresponding to the MACH msgh_id. It may happen that some code needs to parse a MIG generated message, and to recognize the msgh_id. It was common practice to spread #defines anywhere it was needed, without any relation with the MIG .defs. With the -dheader option, Untyped MIG creates an header file that contains all the #defines related to the msgh_ids defined in the .defs (the msgh_ids of the replies are included too).

2.3 Language Extensions

2.3.1 Types

Untyped IPC supports complex types built upon out-of-lines types (either out-of-line data or out-of-line ports).

Example 3

(a) type array_of_array = array[n] of ool_type;
(b) type array_of_array = array[*:n] of ool_type;

The ool_type has to be one of the following:

(c) type ooltype = ^basic_type;
(d) type ooltype = ^array[m] of basic_type;
(e) type ooltype = ^array[*:m] of basic_type;

(f) type ooltype = ^array[] of basic_type;

NOTE1 Basic_type can be either a port or an in-line fixed size scalar type.

NOTE2 A synthetic notation like:

```
type array_of_array = array[*:100] of ^array[] of basic_type;
```

is not allowed. The second part of the expression has to be given a type name.

NOTE3 If the complex types are made out of a variable number of out-of-line regions (case (b)), Untyped MIG will generate an extra argument (type *mach_msg_type_number_t*) to determine at run-time the actual number of out-of-line regions. If the out-of-line regions are variable sized (case (e), (f)), Untyped MIG will generate an extra argument that points to an array of sizes of the out-of-line regions. Appendix A is an example of the case (b)+(e).

NOTE4 IPCflags only apply to the outermost type: e.g.,

Example 4

```
routine my_test(
        target_task     : mach_port_t;
out     par1_in         : array_of_array, Overwrite
)
```

NOTE5 All the IPCflags that can be used in conjunction with out-of-line types apply as well to the complex types built upon out-of-line types.

The use of these types is particularly convenient in those cases where a compound action may take place over a (variable) number of (variable sized) regions of an address space. The use of the Overwrite flag in conjunction with these complex types provides a very powerful and efficient tool to implement interfaces.

2.3.2 Specifications

Additional keywords have been created, to complement the IN and OUT keywords. They have to be used for arguments whose data is in the MACH message trailer: the message sequence number, the security token. Whenever one of the following keywords is defined, the Untyped MIG stubs will contain code to:

- Verify that the type of trailer matches with one of the types of trailers that Untyped MIG is able to handle (at the moment, only MACH_MSG_TRAILER_FORMAT_0). This test is performed regardless of the compile-time TypeCheck switch.
- Verify that the size field of the trailer is consistent with the type and size of the arguments that have to be extracted from the trailer (this test is performed only if TypeCheck is defined).

MsgSeqno — In fact, this is not a new keyword. Its behavior is new: Untyped MIG fetches the data from the field *msgh_seqno* of the trailer (in MACH3 MIG, the sequence number was extracted from the message header). The type of the MsgSeqno argument must be a synonym of the built-in type MACH_MSG_TYPE_INTEGER_32 (the "ctype" being a *mach_port_seqno_t*).

UserSecToken

ServerSecToken — The argument next to these keywords correspond to the field msgh_sender of the trailer. The type of the UserSecToken, ServerSecToken arguments must be a 64 bits type.

Example 5

```
type security_token_t = MACH_MSG_TYPE_INTEGER_64;
                        ;

routine  my_test(
                         target_task      : mach_port_t;
                         par1             : ptr;
ServerSecToken           sectoken         : security_token_t;
out                      opar1            : ptr;
UserSecToken             sectoken         : security_token_t
);
```

UserImpl

ServerImpl — These keywords have been implemented to deal with a multi-element model of trailer. They allow to fetch any data from the trailer, without any information built-in into Untyped MIG: with this support from the MIG Description Language, it is possible to add new elements to the trailer and to update the interfaces to read those elements, without having to change Untyped MIG. There is a caveat: Untyped MIG will not generate any code to TypeCheck these arguments.
The fragment of code:

Example 6

```
type security_token_t = MACH_MSG_TYPE_INTEGER_64;
                        ;

routine  my_test(
                         target_task      : mach_port_t;
                         par1             : ptr;
ServerImpl               sectoken         : security_token_t, msgh_sender;
out                      opar1            : ptr;
UserImpl                 sectoken         : security_token_t, msgh_sender
);
```

shows that the implementor has to provide an extra symbolic name (*msgh_sender*) after the argument name, to match the symbolic name defined in the trailer structure (*mach/message.h*). From Example 6, Untyped MIG will generate the same code as from Example 5 (modulo the code for TypeCheck-ing).

2.3.3 IPC flags

SameCount
: It indicates that the variable sized argument has the same size of the previous argument. Co-bounded arrays are an optimization in that the calling signatures have fewer arguments, the message parsing algorithms and the message itself are smaller.

PhysicalCopy
: It apply to out-of-line arguments only: it requires the argument to be copied with a physical copy, without VM optimizations (e.g., COW). This flag can be useful anytime a predictable timing is required (e.g., in case of a real-time application).

Overwrite
: It apply to out-of-line OUT arguments only. The OUT argument becomes an input argument to the stub, with the address (and size) of the section of address space to be overwritten. When returning from the Untyped MIG stub, the argument will be the same as in input; the size, if present, will be the actual size of the region that has been overwritten.

If the region is too small to contain the data to be overwritten, the kernel will destroy the message and will return MACH_RCV_TOO_LARGE. If MACH_RCV_LARGE is specified at the time the mach_msg_overwrite call is issued, the kernel will not destroy the message, and it will return an header and a list of out-of-line descriptors with the actual size. At the next mach_msg_overwrite call, it will be possible to retrieve the message, once the correct sizes are provided.

RetCode
: This flag has been established to concept messages that match the layout of a reply message. In fact, it may happen that the reply to a RPC request is suppressed (e.g., using MIG_NO_REPLY), and that, later on, an asynchronous message is sent to finish the RPC (and wakeup the client thread).

In this case, the asynchronous message has to perfectly match a reply message: since the sole difference between a request and a reply message is the space reserved for the return code (a *kern_return_t* type), it is possible to instruct Untyped MIG to generate an asynchronous request message that has a *kern_return_t* reserved (e.g., by creating a special .defs that has a *kern_return_t* argument as the first argument).

However, accordingly to the Untyped MIG protocol, the space reserved for the return code is not present in complex reply messages (a complex reply message has an implicit KERN_SUCCESS return code associated

with it; if an error has to be propagated, the minimal *mig_reply_error_t* message is sent instead).
To implement the Untyped MIG protocol correctly, the definition of the asynchronous complex request messages will have a field like:

(a) in return_code : kern_return_t, RetCode;

The flag RetCode will tell Untyped MIG to generate a stub implementing the following algorithm:

```
        if (return_code != KERN_SUCCESS)
(b)             send a mig_reply_error_t structure
        else
(c)             send a complex message without the
                return code field.
```

It is also the case that the very same asynchronous request message may be intercepted as an asynchronous message: the Untyped MIG stub that get such a message will use the reverse algorithm to parse the message:

```
        // a complex message is expected
        if (mig_reply_error_t was rcv'd)
                call target with:
(d)                     target(request_port, <error>, (...)0, (...)0,
        else
                call target with:
(e)                     target(request_port, KERN_SUCCESS, par_1, par_2
```

To be noted that, in case (d), the error condition imposes to receive zero values for par_1, par_2.

A better solution to the problem of creating asynchronous messages that match reply messages is the following:

- Don't use the Untyped MIG error field to carry errors which are set outside the Untyped MIG stubs: the error field is reserved for Untyped MIG internal use only (MIG_TYPE_ERROR, MIG_BAD_ARGUMENTS, etc.)
- Define a new argument in the RPC definition and in the asynchronus request message definition, to be used for end-to-end error notification (Untyped MIG will manage this parameter as a normal argument).
- Line (a) will still be present (to reserve the space). However, it will always be KERN_SUCCESS. (b) and (d) will never be executed. Errors will be notified via the additional argument present in the calling signatures.

2.3.4 Miscellaneous

2.3.4.1 Dispatching among several MIG subsystems

In every server stub, Untyped MIG initializes the variable

```
mig_entry_t <sys>_server_entry;
```

The type mig_entry_t is defined as:

```
typedef struct {
        int start;              // lowest msgh_id
        int end;                // highest msgh_id+1
        mig_routine_t *ptr;     // dispatch table address
        int maxsize;            // largest request message
} mig_entry_t;
```

The scenario is the one where we want to dispatch a MACH message and we have multiple Untyped MIG server stubs. With MACH3 MIG, it is mandatory to make a linear search through the MIG subsystems, using the <sys>_server_routine() utility.
With Untyped MIG, it is possible to build a global dispatcher, which is aware of the characteristics of each Untyped MIG server subsystem (lowest msgh_id, highest msgh_id, routines to be called, maximum size of incoming messages). The global dispatcher will typically use this data to build an hash table, to be preferred to the linear search whenever performances are critical (the MACH kernel uses this technique to dispatch kernel services).

2.3.4.2 MIG-generated prototypes

Untyped MIG is able to produce code that is effective on environments where the use of prototypes is as strict as possible.

Client side	Use the -header to specify the header file that has the prototypes to be used in the client environment.
Server side	Use the -sheader to specify the header file that has the prototypes to be used in the server environment.

In both cases, the MIG-generated header file will be automatically included by the stub files. The MIG-generated header files will also be manually included by the code that calls the stub (at the client side) and by the code that implements the target routine (at the server side). The pervasive use of MIG-generated prototypes provides additional guarantees that a MIG_based interface is implemented and referenced correctly.

2.3.4.3 Identification of MIG-generated modules

To minimize the errors coming from source management control and build procedures (am I running the "right" MIG ?), the MIG-generated stubs have an identification banner, at the beginning:

```
/*
 * IDENTIFICATION:
 * stub generated Wed May 19 10:56:29 1993
 * with a MiG generated Tue May 18 18:39:04 EDT 1993 by travos@roast
 * OPTIONS:
 *     KernelServer
 */
```

3. Deletions

3.1 Types

The expressions:

(ipc-typename, size [, IPC-flags])

are not allowed any longer. The only way to introduce a new type is the following:

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

The use of polymorphic types has been restricted to the use of MACH3 ports: the built-in type MACH_MSG_TYPE_POLYMORPHIC is considered now to be a port type (like MACH_MSG_TYPE_COPY_SEND). It is not possible anymore to declare an argument that can be either a port or a scalar value: the argument must be a port, and the type (e.g., MACH_MSG_TYPE_COPY_SEND) will be specified at run-time as an extra argument passed to the stub.

In-line unbounded arrays, defined in the form:

(a) array[] of comp-type-desc

are always sent out-of-line. The cut-off value checked at run-time (if the size was less than 2048 bytes MACH3 MIG used in-line, if greater or equal out-of-line) has been suppressed. In other words, Untyped MIG uses a cut-off value of 0.
The expression above is equivalent to the out-of-line declaration for unbounded arrays:

(b) ^array[] of comp-type-desc

(the only exception being that with (a) IN arguments will be automatically de-allocated by the server stub, upon return from the target routine. With (b), it will be up to the target routine to de-allocate the dynamically allocated space).

3.2 IPC flags

The flags IsLong, IsNotLong are not recognized by the Untyped MIG.

The flag ServerCopy has also been pruned: its use was strictly related to the presence of a non-zero cut-off value for in-line unbounded arrays.

4. Migration Guide

To migrate a MIG Description File from MACH3 MIG to Untyped MIG, few rules need to be followed. They are the direct consequence of the deletions listed in the previous section.

- The expressions like:

  (ipc-typename, size [, IPC-flags])

  need to be converted to:

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

  A typical example is the following:

  Example 7

  ```
  #if     Untyped_MIG
  type kernel_version_t    = c_string[*:512];
  type kernel_boot_info_t = c_string[*:4096];
  #else  /* !Untyped_MIG */
  type kernel_version_t    = (MACH_MSG_TYPE_STRING, 512*8);
  type kernel_boot_info_t = (MACH_MSG_TYPE_STRING, 4096*8);
  #endif /* !Untyped_MIG */
  ```

- If polymorphic types are used, it has to be checked whether the code that interacts with the MIG stubs handles the polymorphic argument as a scalar type: if this happens, there will be errors at run-time.

- In theory, the use of in-line unbounded arrays does not require any change. However, there are several side effects introduced by the fact that Untyped IPC uses out-of-line only for this kind of arguments:

  - If the in-line unbounded array is an OUT argument, the code that calls the MIG stub (user side) can be optimized (the usual check to see whether the argument was sent in-line or out-of-line is useless).

  - If the in-line unbounded array is an OUT argument with the CountInOut flag, the CountInOut flag has to be removed (CountInOut is meaningless when the argument is out-of-line). As in the previous paragraph, the code that interfaces the MIG stubs can be polished, and some useless code can be removed.

- The flags IsLong and IsNotLong have to be removed. There is no impact on the code that interfaces with the MIG stubs.

- The flag ServerCopy has to be removed. There is an impact on the code of the target routine that the MIG server stub invokes: in fact, the calling signature is different (ServerCopy introduces an ancillary argument).
- There will be migration problems anytime asynchronous messages try to play the role of reply messages (e.g., the user makes an RPC - the reply message is not sent (MIG_NO_REPLY) - an asynchronous message is generated later on to finish the RPC). In fact, this operation requires an deep knowledge of the layout of a Reply message. The flag RetCode has been established to address these problems (see the RetCode section).

5. APPENDIX A

Example 8

A client application interacts with a server application by sending and receiving a variable number (0<= n <= MAX_OOL) of variable arrays of integers (0 <= m <= SIZE_OOL). Since the data coming from the server represents a fresh copy of the data held by the client, the client overwrites its own copy of the data.

The MIG Description Language file is:

```
subsystem test 1200;



import  "my1_ctypes.h";

type ptr = ^array[*:SIZE_OOL] of int;
type complex = array[*:MAX_OOL] of ptr;

routine  my_test(
                target_task     : mach_port_t;
                par1_in         : complex;
        out     par2_out        : complex, overwrite);
```

The import directive will be translated by MIG into a C #include of the header file that contains the C types for the complex MIG types:

```
typedef int *     ptr;
typedef int **    complex;
```

The code of the client that is related to the RPC is:

```
int
main(
	int	argc,
	char	**argv)
{
	ptr		in_array_of_pointers[MAX_OOL];
	ptr		out_array_of_pointers[MAX_OOL];
	ptr		fixed_array_of_pointers[MAX_OOL];
	complex			in_ptrptr;
	complex			out_ptrptr;
	mach_msg_type_number_t in_array_of_counters[MAX_OOL];
	mach_msg_type_number_t out_array_of_counters[MAX_OOL];
	mach_msg_type_number_t fixed_array_of_counters[MAX_OOL];
	mach_msg_type_number_t in_ool_count, out_ool_count;
	int retcode, i;

	/* beginning of initialization */
	for (i = 0; i < MAX_OOL; i++) {
		in_array_of_counters[i] = SIZE_OOL;
		in_array_of_pointers[i] = (ptr)malloc(SIZE_OOL * sizeof(int));
		fixed_array_of_counters[i] = SIZE_OOL;
		fixed_array_of_pointers[i] = (ptr)malloc(SIZE_OOL * sizeof(int));
	}

	in_ptrptr = &in_array_of_pointers[0];
	out_ptrptr = &out_array_of_pointers[0];
	/* end of initialization */

start_loop:
	/*
	 * To be initialized at each iteration:
	 *	1. the number of regions to be sent (in_ool_count)
	 *	2. the size of each region (in_array_of_counters[])
	 */
	to_be_sent(&in_ool_count, in_array_of_counters);

	/*
	 *	3. the number or regions we're ready to get (maximum)
	 */
	out_ool_count = MAX_OOL;

	/*
	 *	4. the overwrite template
	 */
	for (i = 0; i < MAX_OOL; i++)  {
	   out_array_of_counters[i] = fixed_array_of_counters[i];
```

```
    out_array_of_pointers[i] = fixed_array_of_pointers[i];
}



/* remote procedure call */
retcode = my_test(serv_port,
    in_ptrptr, &in_array_of_counters[0], in_ool_count,
    out_ptrptr, &out_array_of_counters[0], &out_ool_count
);

if (retcode != KERN_SUCCESS) {
    printf("Result from user side stub is %d (%x)", retcode, retcode);
    exit(1);
}
for (i = 0; i < out_ool_count; i++)
    if (out_array_of_pointers[i] != fixed_array_of_pointers[i]) {
            printf("ERROR: New memory has been allocated! <%d> %x != %x", i,
               out_array_of_pointers[i], fixed_array_of_pointers[i]);
            exit(1);
    }
...
```

The code of the server that is related to the RPC is:

```
kern_return_t
my_test(
        mach_port_t server_port,
        complex in_ptrs,
        mach_msg_type_number_t *in_counters,
        mach_msg_type_number_t in_cnt,
        complex out_ptrs,
        mach_msg_type_number_t *out_counters,
        mach_msg_type_number_t *out_cnt)
{
        /*
         * server discipline
         */
         ...
}
```

Bibliography

[Loepere92] Loepere, K., MACH3 Server Writer's Guide Open Software Foundation and Carnegie Mellon University, 1992.

[Loepere93] Loepere, K., MACH3 Server Writer's Guide - Draft Proposed Specification, Open Software Foundation and Carnegie Mellon University, 1993.

[Loepere93a] Loepere, K., MACH3 Kernel Interfaces - Draft Proposed Specification, Open Software Foundation and Carnegie Mellon University, 1993.

[Kong91] Kong, M., Network Computing Architecture, version 2.0, Cooperative Computing Division - Hewlett-Packard Company, 1991.

[DCE91] DCE Application Development Guide, Open Software Foundation.

[Travos93] Travostino, F., Untyped MIG: The Protocol, Open Software Foundation - Research Institute, 1993.

CONTENTS