**OSF Research Institute**

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

# Untyped MIG: The Protocol

**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

This document complements [Travos93] with the details of the algorithms used in the Untyped MIG stubs. Naming and conventions are consistent with [Travos93].

If compared to MACH3 MIG, Untyped MIG exploits two additional functionalities:

- It organizes the data to be marshalled/unmarshalled into the kernel processed section and the untyped section of the MACH message. Whereas the first section has to be compliant to the Kernel Interface Specification, Untyped MIG is entirely responsible for handling the data of the latter section, and providing a strategy for end-to-end consistency.
- It performs type-conversions accordingly to the Network Data Representation (NDR) protocol.

The new set of features added to Untyped MIG suggested us to introduce the notion of MIG protocol. In general, the details of the Untyped MIG protocol are hidden to the developer who uses the combination of client and server stubs generated by MIG. However, a scenario like the following one is realistic:

- The user application uses his own stub generator (eventually Raw MACH IPC), but it wants to inter-operate with Untyped MIG generated stubs, either as a client or as a server. Thus, the user application (or the custom stub generator) need to be aware of the Untyped MIG protocol.

The chapters follow the order of the fields in the message.

# 2. Header

## 2.1 Simple/Complex Messages

A MACH message is defined (and tagged) as simple when there is no need for a kernel processed section (thus, no ports, no out-of-line). Such an attribute will grant to the MACH message the most efficient path in the kernel processing.
With a quick inspection of the MIG language source file, it is possible to determine whether a request or reply message is simple or not.

- The message will be complex if at least one of these cases apply:
  - An argument is defined to be a port (besides the request and reply ports).
  - An argument is defined as out-of-line.
  - An argument is a variable unbounded array.

- In the Untyped IPC there is only one case where the message may change from complex to simple: the error case (RetCode not zero). Vice versa, in the MACH3 IPC there are two more cases on which the nature of the message is changed runtime from simple to complex:
  - Polymorphic types: at run-time, MACH3 MIG users could describe an argument to be an integer (simple message) as well as a port (complex message). In Untyped IPC, polymorphic types have been restricted to port rights (e.g., users can specify at run-time whether they want a MOVE_RECEIVE or a MOVE_SEND).
  - Unbounded in-line arrays: at run-time, the MACH3 MIG stub decides upon the size of an argument (cut-off threshold at 2048 bytes) whether the message has to be sent in-line (simple message) or out-of-line (complex message). In Untyped MIG, unbounded in-line arrays are always sent out-of-line, due to the more efficient implementation of the out-of-line semantics.

  Reply messages carrying error messages (RetCode not zero) are always simple, regardless of what we would expect: this case will be detected by the following code in the stub at the client side:

```
if (message is simple &&
        ((mig_reply_error_t *)pointer)->RetCode != KERN_SUCCESS)
{
        // an error has been reported
}
```

*(see the RetCode paragraph for further details on the RetCode field and on the mig_reply_error_t structure).*

## 2.2 Reply Port

The reply port field of the header determines whether the client will receive a reply to the request message: if reply port is set to be MACH_PORT_NULL, it will be an asynchronous communication (in MIG terminology, a SimpleRoutine).
If reply port is set to be different from MACH_PORT_NULL, there will be a reply message, unless the server stub (or the target routine) set the RetCode to the value MIG_NO_REPLY.

The code generated by Untyped MIG for the client stub makes an optimized use of the mach_reply_port() trap and of the SEND_ONCE disposition: in the optimal cases, mach_reply_port() is avoided, and the same port right (defined as a global variable in libmach.a) is recycled (same as in MACH3 MIG).

The code of the client stub has also to be able to handle the case where a notification is received instead of a reply message.

## 2.3 Message ID

Unlike other stub generators (e.g. see DCE RPC), MIG does not rely upon a generator of Universal Unique Identifiers. Therefore, it will be up to the implementor to survey the message IDs in use on the system and pick up a fragment of the large and sparse ID space.

As an example, these are the message IDs currently in use for the MACH kernel services:

| | |
|---|---|
| (2000, 2100) | mach.defs |
| (3200, 3218) | mach_port.defs |
| (2600, 2678) | mach_host.defs |
| (2800, 2812) | device.defs |
| (2200, 2209) | device_pager.defs |
| (3000, 3018) | mach_debug.defs |
| (555000, 555012) | mach_norma.defs |
| (556000, 556010) | norma_internal.defs |
| (923000, 923015) | proxy.defs |
| (923000, 923015) | mach_i386.defs |

While selecting his own range of message IDs, the developer has to pay attention to the fact that the message ID in the reply message is the message ID of the request incremented of 100.

If the client stub detects a wrong message ID in the reply message, it is supposed to return to its caller a MIG_REPLY_MISMATCH. If, more precisely, the wrong message ID in the reply message happens to be a message ID used for the notifications (i.e. MACH_NOTIFY_SEND_ONCE), the return code could indicate that the server is not alive any longer (MIG_SERVER_DIED).

# 3. Kernel Processed Section

The entries of the kernel processed section describe (see [Loepere93a], mach_msg_descriptor):

- Ports (MACH_MSG_PORT_DESCRIPTOR)
- Out-of-line regions (MACH_MSG_OOL_DESCRIPTOR)
- Out-of-line ports (MACH_MSG_OOL_PORTS_DESCRIPTOR)

It is possible to determine how many entries are present in a kernel processed section, and therefore the size of the kernel processed section, from the field *msgh_descriptor_count* in the extension of the message header (see [Loepere93a], mach_msg_header).

It is also possible to get the number of entries from the MIG Description Language. However, we do not recommend this solution because:

- It is not always easy to figure out how many kernel processed entries an argument takes.
- The method of reading *msgh_descriptor_count* is more robust against small changes to the MIG protocol that may take place in a near future, such as the ones described hereafter:
  - The message will be compressed, before being sent, to skip unused kernel processed entries (e.g., in case of variable size arguments).
  - The kernel has the freedom to change the number of kernel processed entries, through addition or removal of entries (e.g., piggy-backing of attributes, compound operations, etc.).

The kernel processed entries are organized, within the kernel processed section of the MACH message, in the same order as the corresponding arguments are defined in the MIG Description Language: in particular, for the request message the order is the one of the In and InOut arguments, for the reply message it is the one of the InOut and Out arguments.

Besides the order, it is also important to know how many kernel processed entries are occupied by an argument. In fact, different declarations may range from the minimum of one kernel processed entry to an unlimited maximum.

- A port, an out-of-line region and an out-of-line array of ports take one entry.
- Complex types, built upon the above types, will occupy as many entries as the ratio between the size of the complex type and the size of the building type. For example, an in-line array of 10 ports will occupy 10 entries; similarly, an array of 10 pointers to out-of-line regions will occupy 10 entries.

  These complex types are in fact implemented by enumerating the types that they are made of.

  Since the MIG Description Language allows to describe variable sized arguments, a policy needed to be established for arguments whose size is specified only at run-time.

  - The maximum number of kernel processed entries for a variable sized argument is determined from the MIG Description Language, and these entries are statically allocated in the message.
  - At runtime, only the exact number of used kernel processed entries is marshalled/unmarshalled. Such a number can be retrieved in one of the two ways:
    1. From the arguments passed to the stub, if we are packing a request message at the client side or we are packing a reply message at the server side.
    2. From the untyped section of the message, if we are un-packing a reply message at the client side or we are un-packing a request message at the server side. In fact, within the untyped section of the message there is a field (*mach_msg_type_number_t*) with the actual number of kernel processed entries used for a complex argument: the position of this field is the one that the whole complex argument would have occupied in the untyped section, if it the argument was untyped.

    The unused kernel processed entries need to be properly initialized, to let the kernel know that it has to skip these entries. In order to set up kernel no-ops:

    - MACH_MSG_PORT_DESCRIPTOR entries must have the *name* field equal to MACH_PORT_NULL.
    - MACH_MSG_OOL_DESCRIPTOR entries must have the *size* field equal to 0.
    - MACH_MSG_OOL_PORTS_DESCRIPTOR entries must have the *count* field equal to 0.

A set of examples (MIG Description Language fragments, with a short description of what happens) follows, to recall some typical behaviors:

```
...
par1     : mach_port_t;
...
```

The argument par1 requires a single kernel processed entry (of the type MACH_MSG_PORT_DESCRIPTOR).

```
...
```

```
par2     : complex = array[10] of mach_port_t;
...
```

The argument par2 requires 10 kernel processed entries (of the type MACH_MSG_PORT_DESCRIPTOR).

```
...
par3     : complex = array[*:10] of mach_port_t;
...
```

The argument par3 requires 10 kernel processed entries (of the type MACH_MSG_PORT_DESCRIPTOR). In addition, its run-time count is inserted into the untyped section (a *mach_msg_type_number_t*, 0<= m <= 10).

```
...
par4     : complex = ^array[10] of mach_port_t;
...
```

The argument par4 requires a single kernel processed entry (of the type MACH_MSG_OOL_PORTS_DESCRIPTOR).

```
...
par5     : ool = ^array[10] of int;
...
```

The argument par5 requires a single kernel processed entry (of the type MACH_MSG_OOL_DESCRIPTOR). The field *size* of this entry will be set to (10 * sizeof(int)).

```
...
par6     : var_ool = ^array[*:10] of int;
...
```

The argument par6 requires a single kernel processed entry (of the type MACH_MSG_OOL_DESCRIPTOR). The field *size* of this entry will be set to the value y, with 0 <= y <= (10 * sizeof(int)). Its run-time count is also inserted into the untyped section (a *mach_msg_type_number_t*, 0 <= m <= 10).

```
...
type ptr = ^array[*:10] of int;
...
par7     : ool_ool = array[*:4] of ptr;
...
```

Par7 is a variable sized array of variable sized out-of-line regions. It takes 4 kernel processed entries (of the type MACH_MSG_OOL_DESCRIPTOR). In the untyped section we will find, in the order, the number of kernel processed entries (0 <= x <= 4) and the runtime counts related to the subordinate arrays (they will be as many as the number of the kernel processed entries; they will be 0 <= y <= 10).

# 4. Untyped Section

## 4.1 NDR format label

NDR is the protocol used to encode the untyped data. MIG implements a subset of the NDR protocol, which is described in its integrity in NCA [Kong91].

To describe in a uniform way the data representation used on a machine (e.g., whether it is a Big Endian or a Little Endian), NDR rely upon a format label, consisting of 4 bytes (the layout of such a format label is the same as in NCA). In addition, MIG expands the NDR format label to 8 bytes, indicating the version numbers of the MIG protocol, of the interface and of the NDR protocol itself. The NDR format label is always at the beginning of the untyped section of any message; if there is no reason for having untyped data in a particular message (e.g., a null message for synchronization, a complex message with a port), the NDR format label is omitted too.

NDR format label:

```
typedef struct {
        unsigned char   mig_vers;
        unsigned char   if_vers;
        unsigned char   reserved1;
        unsigned char   mig_encoding;
        unsigned char   int_rep;        // standard NDR
        unsigned char   char_rep;       // standard NDR
        unsigned char   float_rep;      // standard NDR
        unsigned char   reserved2;      // standard NDR
} NDR_record_t;
```

mig_vers — The version number of the MIG protocol.

if_vers — The version number of an interface. It allows to change an interface without changing the message ID that corresponds to that interface. The per-interface version number is indicated in the .defs. The stubs will make sure that the incoming packets match the version numbers, when specified (and when TypeCheck is turned on).
*(not available yet)*

mig_encoding — It indicates the type and version number of the protocol used to encode untyped data. At the moment, only one protocol is recognized, NDR 2.0:

```
#define NDR_PROTOCOL_2_0        0
```

int_rep — Integer representation:

```
#define NDR_INT_BIG_ENDIAN      0
#define NDR_INT_LITTLE_ENDIAN   1
```

char_rep Char representation:

```
#define NDR_CHAR_ASCII     0
#define NDR_CHAR_EBCDIC    1
```

float_rep Floating point representation:

```
#define NDR_FLOAT_IEEE     0
#define NDR_FLOAT_VAX      1
#define NDR_FLOAT_CRAY     2
#define NDR_FLOAT_IBM      3
```

If Untyped MIG moves to another encoding schema, the format label implemented by Untyped MIG will look like:

```
typedef struct {
        unsigned char   mig_vers;     // mig_vers++
        unsigned char   if_vers;
        unsigned char   reserved1;
        unsigned char   mig_encoding; // xyz protocol
         -----------------
        |
        | xyz format label ...
        |
         -----------------
} xyz_record_t;
```

## 4.2 Return Code

The RetCode field (*kern_return_t*) is present on any reply message that is a simple MACH message too. Complex reply messages do not reserve any space to be used as RetCode: if Untyped MIG has to notify its peer that an error has occurred, it will switch from the complex reply to the minimal *mig_reply_error_t* structure.

RetCode can take one of the following values:

- MIG_BAD_ID, if the msgh_id selected did not match with any service.
- MIG_BAD_ARGUMENTS, if the checking of the request message header has found some errors.
- MIG_TYPE_ERROR, if the type-checking of the request message typed section has found some errors.
- MIG_NO_REPLY, to tell the server mainloop to discard the reply (the effect of such a code is local to the server side. The client stub will never receive the Reply message).
- The return code from the target routine. In order not to enter into the error handling logic, the target routine has to return a null return value (e.g., KERN_SUCCESS).

To easily access the RetCode, the *mig_reply_error_t* is provided: it is intended to be the minimal Reply message structure for errors:

```
typedef struct {
        mach_msg_header_t     Head;
        NDR_record_t          NDR;
        kern_return_t         RetCode;
} mig_reply_error_t;
```

## 4.3 Data

The arguments in the untyped section are organized in the same order as the untyped arguments (no ports, no out-of-line) are defined in the description of the interface: in particular, for the request message the order is the one of the In and InOut arguments (with the inclusion of the Out arguments marked with the CountInOut keyword), for the reply message it is the one of the Out arguments. In both cases, they come after the NDR record and the RetCode (if present).

In the untyped section of the message there are also the variables (type *mach_msg_type_number_t*) that carry the actual size of variable sized arguments using kernel processed entries. The position of each of these variables is the one that the whole complex argument would have occupied in the untyped section, if it the argument was untyped.

The subset of NDR rules used by Untyped MIG to encode the data in the Untyped Section is described in detail in [Loepere93a], Appendix A.

# 5. Trailer

Since the implicit data present in the Trailer (e.g., message sequence number, security token) is always generated by the kernel, there is no need for Untyped MIG to use any encoding schema. Thus, the rules to access the Trailer data are the same as in the Raw MACH IPC case.

# Bibliography

[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: What Has Changed and Migration Guide, Open Software Foundation - Research Institute, 1993.

CONTENTS