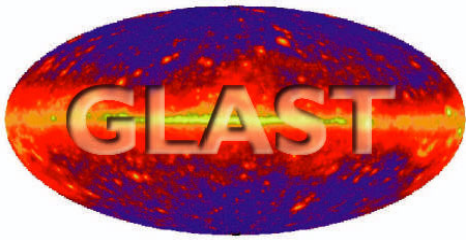

LAT Flight Software

MSG Manual

Type:	User Manual
Version:	V1-1-0
Author:	A.P.Waite
Created:	27 October 2003
Updated:	28 October 2003
Printed:	30 October 2003

Manual for the MSG system.

Contents

Figures

Tables

0 Preface

The message utility (MSG) is not particularly original work. It owes around 70% of its heritage to the VMS `LIB$signal` system (though it should be made clear that MSG can only do about 10% of what `LIB$signal` could do). There are also influences from the SLD experiment's `SLDERROR` system and even a little of `CMlog`.

MSG seeks to provide a uniform method for dealing with what in VMS were euphemistically called "conditions". Most of the time that means "errors". Error reporting and structured error handling have always been thorny issues in computing. Many languages now offer extensions precisely to help with error handling (the "throw-catch" syntaxes). I can't rewrite the compiler to help users but MSG can provide some value added.

Code developers wishing to use the message system will have to learn new techniques at both package build time and at run time. At package build time the developer must:

- Prepare special message files (section 1).
- Introduce these message files into the package requirements document (section 3).

At run time, the developer will have to understand:

- How to get a message "processed" using the `MSG_signal()` call (section 2.0).
- How to deal with the message codes the developer's code will receive (section 2.1).
- How to start up and configure the message system (section 4).

1 Anatomy Of A `.msg` File

At the heart of the MSG system is a method to associate a unique 32-bit number with a parameterizable text string. Computers are very good at processing numbers whereas humans are better at processing text. The associations are constructed in special format files with the `.msg` extension. A MSG provided utility called `msg2src` can parse `.msg` files to produce a `.c` source file and a `.h` header file.

The `.c` source file is constructed as a static constructor and a static destructor which run automatically. The constructor's purpose is to hang the processed contents of the `.msg` file on a singly linked list for access at run time. The destructor's purpose is to remove it.

The `.h` file is simply a list of definitions associating a moderately mnemonic string with a hex number. This is how message codes are "advertised" to other packages. The `.h` file typically ends up in a package's export directory.

1.0 Example `.msg` File From MSG Itself

To make this more real, here is an annotated `.msg` file called `MSG_msgs.msg` followed by its derived files:

Yes indeed the MSG package uses the MSG system to report its own messages. All very recursive and confuses the hell out of me most of the time.

```
0  # ------------------------------------------------------------------------
   # CVS $Id: MSG_msgs.msg,v 1.2 2003/10/22 00:26:04 apw Exp $
   #
   # Messages associated with the MSG system itself.
   # ------------------------------------------------------------------------
1
2  --FACILITY MSG 254

3  SUCCESS  S "Success"

   # ------------------------------------------------------------------------
   # Alternate messages when MSG_signal() receives a nonsense message code.
   # ------------------------------------------------------------------------
   UNIXGOOD S "Unrecognised message 0x%08x (possibly unix success)"
   UNIXEROR E "Unrecognised message 0x%08x (possibly unix error)"
   UNKNOWNS S "Unrecognised message 0x%08x (success)"
   UNKNOWNI I "Unrecognised message 0x%08x (information)"
   UNKNOWNW W "Unrecognised message 0x%08x (warning)"
   UNKNOWNE E "Unrecognised message 0x%08x (error)"

   # ------------------------------------------------------------------------
   # Messages associated with the control of the message system itself.  Note
   # that many of these messages cannot be recorded because they occur while
   # the message system is in no fit state to do so.  They are therefore very
   # simple messages that do not take advantage of the "printf" quality of the
   # formatting string.
   # ------------------------------------------------------------------------
   BADSTATE E "Request illegal in current state of message"
   FORKFAIL E "Forking the message task failed"
   FPAIFAIL E "FPA initialization failed"
   FPAGFAIL E "FPA_get() failed during start"
   OUTNTFND E "Target of MSG_deletOutputRtn() not found"
   PKT2SMAL E "Initialization parameter packet length too small"
   PKT2FEW  E "Initialization parameter packet count too small"

   # ------------------------------------------------------------------------
   # Other assorted messages
   # ------------------------------------------------------------------------
   ALOCFAIL E "Could not allocate %d bytes"
   TNM2LONG E "A task name is restricted to <= %d letters"

   # ------------------------------------------------------------------------
   # Messages used to announce begin blackout and end blackout
   # ------------------------------------------------------------------------
   GAPBEGIN E "Message blackout starts"
   GAPEND   E "Message blackout ends"
```

Figure 1 The file `MSG_msgs.msg`

Notes:

0. Lines with a # character in the first column are comment lines.
1. Blank lines can be used to improve legibility.
2. The first parseable line in a `.msg` file must always define a facility name and number. The word “facility” is taken directly from the VMS implementation. I retained it because a `.msg` file does not necessarily map exactly to either a CMX package or a CMX constituent.

The line must contain exactly three tokens:

- The first token must be exactly `--FACILITY` (case sensitive).
- The second token is the facility name and must be one to four characters long. The characters must be drawn from the set of upper case alphabetic characters, digits, or the underscore character. The first character must either upper case alphabetic or the underscore character.
- The third token is the facility number. This must be in the range 0-255. Some of these numbers are already allocated and the allocation of the remainder will be dealt with in section 1.2.

3. All remaining parseable lines in the file must be message defining lines. A message line consists of three tokens:
   - A mnemonic name three to eight characters long. The rules for forming this name are otherwise similar to forming a facility name.
   - A single character drawn from the set `[SIWE]`, designed to indicate the message severity. `S` stands for success, `I` for information, `W` for warning and `E` for error.
   - A compound string that looks just like a `printf` formatting string (mostly because that is what it is!). The only constraint is that special escape characters (newline, tab, bell, …) are not allowed.

Running `msg2src` on this `.msg` file produces the following `.c` and `.h` files (please note that I've dropped the font size on the `.c` file to try to squeeze it on the page, but some of the lines have still wrapped). The majority of the `.h` file is dedicated to producing legible Doxygen documentation. The real meat of the `.h` file is at the end.

```
     /*---------------------------------------------------------------------*//*!
     \file  MSG_msgs.c
     \brief Constructor/destructor routines for message facility \b MSG (ID: \c 254, \c 0xfe)
     \warning Machine generated code - NEVER edit by hand
     *//*---------------------------------------------------------------------*/

     #include <stdio.h>
     #include "MSG/MSG_pubdefs.h"

     #ifdef __cplusplus
     extern "C" {
     #endif

     /*---------------------------------------------------------------------*//*!
     \var   MSG_MsgList_MSG
     \brief Formatting strings for message facility MSG
     *//*---------------------------------------------------------------------*/
     static const struct _MSG_MsgList MSG_MsgList_MSG[18] =
     {
         { 0x0000ccea, 0, 7,  7, 0, 0x00000000, "SUCCESS",  "Success" },
         { 0x0009ab8e, 0, 8, 51, 1, 0x00000001, "UNIXGOOD", "Unrecognised message 0x%08x (possibly unix
success)" },
         { 0x0009a2f5, 3, 8, 49, 1, 0x00000001, "UNIXEROR", "Unrecognised message 0x%08x (possibly unix
error)" },
         { 0x0009de96, 0, 8, 37, 1, 0x00000001, "UNKNOWNS", "Unrecognised message 0x%08x (success)" },
         { 0x0009deac, 2, 8, 41, 1, 0x00000001, "UNKNOWNI", "Unrecognised message 0x%08x (information)"
},
         { 0x0009dec6, 1, 8, 37, 1, 0x00000001, "UNKNOWNW", "Unrecognised message 0x%08x (warning)" },
         { 0x0009dea2, 3, 8, 35, 1, 0x00000001, "UNKNOWNE", "Unrecognised message 0x%08x (error)" },
         { 0x000c9bca, 3, 8, 43, 0, 0x00000000, "BADSTATE", "Request illegal in current state of
message" },
         { 0x000a8c1c, 3, 8, 31, 0, 0x00000000, "FORKFAIL", "Forking the message task failed" },
         { 0x000b9e1c, 3, 8, 25, 0, 0x00000000, "FPAIFAIL", "FPA initialization failed" },
         { 0x000bb01c, 3, 8, 29, 0, 0x00000000, "FPAGFAIL", "FPA_get() failed during start" },
         { 0x00086c3e, 3, 8, 40, 0, 0x00000000, "OUTNTFND", "Target of MSG_deletOutputRtn() not found"
},
         { 0x000d24e8, 3, 8, 48, 0, 0x00000000, "PKT2SMAL", "Initialization parameter packet length too
small" },
         { 0x00034b1a, 3, 7, 47, 0, 0x00000000, "PKT2FEW",  "Initialization parameter packet count too
small" },
         { 0x00050e1c, 3, 8, 27, 1, 0x00000001, "ALOCFAIL", "Could not allocate %d bytes" },
         { 0x000252ee, 3, 8, 42, 1, 0x00000001, "TNM2LONG", "A task name is restricted to <= %d letters"
},
         { 0x000c438d, 3, 8, 23, 0, 0x00000000, "GAPBEGIN", "Message blackout starts" },
         { 0x0000c22e, 3, 6, 21, 0, 0x00000000, "GAPEND",   "Message blackout ends" }
     };

     /*---------------------------------------------------------------------*//*!
     \var   MSG_FacList_MSG
     \brief Facility header structure for message facility MSG
     *//*---------------------------------------------------------------------*/
     static struct _MSG_FacList MSG_FacList_MSG =
     {
         NULL, 254, 3,   18, &MSG_MsgList_MSG[0], "MSG"
     };
```

```
void _GLOBAL__I_MSG_insertFacility_MSG() __attribute__ ((constructor));
void _GLOBAL__D_MSG_removeFacility_MSG() __attribute__ ((destructor));

/*---------------------------------------------------------------------------*//*!
\fn    void _GLOBAL__I_MSG_insertFacility_MSG
\brief Static constructor to insert message facility MSG

Static constructor to insert facility MSG into the message facility list
*//*---------------------------------------------------------------------------*/
void _GLOBAL__I_MSG_insertFacility_MSG()
{
    MSG_insertFacility( &MSG_FacList_MSG );
    return;
}

/*---------------------------------------------------------------------------*//*!
\fn    void _GLOBAL__D_MSG_removeFacility_MSG
\brief Static destructor to remove message facility MSG

Static destructor to remove facility MSG from the message facility list
*//*---------------------------------------------------------------------------*/
void _GLOBAL__D_MSG_removeFacility_MSG()
{
    MSG_removeFacility( &MSG_FacList_MSG );
    return;
}

#ifdef __cplusplus
}
#endif
```

Figure 2 The file `MSG_msgs.c`

```
    /*-----------------------------------------------------------------------*//*!
    \file  MSG_msgs.h
    \brief Message definitions for message facility \b MSG (ID: \c 254, \c 0xfe)
    \warning Machine generated code - NEVER edit by hand

    MSG_msgs.h provides all the \c #define statements for the codes in message
    facility \b MSG.  The following table provides the equivalences between
    the \c #define name, the \c #define value and the associated formatting
    string.  For the complete \b MSG message facility data definition, see
    #MSG_FacList_MSG and #MSG_MsgList_MSG.

    <table>
    <tr>
      <td>Name</td>
      <td>Value</td>
      <td>Associated formatting string</td>
    </tr>
    <tr>
      <td><tt>MSG_SUCCESS</tt></td>
      <td><tt>0x7f0333a8</tt></td>
      <td><tt>"Success"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_UNIXGOOD</tt></td>
      <td><tt>0x7f26ae38</tt></td>
      <td><tt>"Unrecognised message 0x%08x (possibly unix success)"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_UNIXEROR</tt></td>
      <td><tt>0x7f268bd7</tt></td>
      <td><tt>"Unrecognised message 0x%08x (possibly unix error)"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_UNKNOWNS</tt></td>
      <td><tt>0x7f277a58</tt></td>
      <td><tt>"Unrecognised message 0x%08x (success)"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_UNKNOWNI</tt></td>
      <td><tt>0x7f277ab2</tt></td>
      <td><tt>"Unrecognised message 0x%08x (information)"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_UNKNOWNW</tt></td>
      <td><tt>0x7f277b19</tt></td>
      <td><tt>"Unrecognised message 0x%08x (warning)"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_UNKNOWNE</tt></td>
      <td><tt>0x7f277a8b</tt></td>
      <td><tt>"Unrecognised message 0x%08x (error)"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_BADSTATE</tt></td>
      <td><tt>0x7f326f2b</tt></td>
      <td><tt>"Request illegal in current state of message"</tt></td>
```

```
    </tr>
    <tr>
      <td><tt>MSG_FORKFAIL</tt></td>
      <td><tt>0x7f2a3073</tt></td>
      <td><tt>"Forking the message task failed"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_FPAIFAIL</tt></td>
      <td><tt>0x7f2e7873</tt></td>
      <td><tt>"FPA initialization failed"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_FPAGFAIL</tt></td>
      <td><tt>0x7f2ec073</tt></td>
      <td><tt>"FPA_get() failed during start"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_OUTNTFND</tt></td>
      <td><tt>0x7f21b0fb</tt></td>
      <td><tt>"Target of MSG_deletOutputRtn() not found"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_PKT2SMAL</tt></td>
      <td><tt>0x7f3493a3</tt></td>
      <td><tt>"Initialization parameter packet length too small"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_PKT2FEW</tt></td>
      <td><tt>0x7f0d2c6b</tt></td>
      <td><tt>"Initialization parameter packet count too small"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_ALOCFAIL</tt></td>
      <td><tt>0x7f143873</tt></td>
      <td><tt>"Could not allocate %d bytes"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_TNM2LONG</tt></td>
      <td><tt>0x7f094bbb</tt></td>
      <td><tt>"A task name is restricted to <= %d letters"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_GAPBEGIN</tt></td>
      <td><tt>0x7f310e37</tt></td>
      <td><tt>"Message blackout starts"</tt></td>
    </tr>
    <tr>
      <td><tt>MSG_GAPEND</tt></td>
      <td><tt>0x7f0308bb</tt></td>
      <td><tt>"Message blackout ends"</tt></td>
    </tr>
    </table>
    *//*-------------------------------------------------------------------------*/

    #ifndef MSG_MSGS_H
    #define MSG_MSGS_H
```

```
#ifndef CMX_DOXYGEN

#define  MSG_SUCCESS          0x7f0333a8
#define  MSG_UNIXGOOD         0x7f26ae38
#define  MSG_UNIXEROR         0x7f268bd7
#define  MSG_UNKNOWNS         0x7f277a58
#define  MSG_UNKNOWNI         0x7f277ab2
#define  MSG_UNKNOWNW         0x7f277b19
#define  MSG_UNKNOWNE         0x7f277a8b
#define  MSG_BADSTATE         0x7f326f2b
#define  MSG_FORKFAIL         0x7f2a3073
#define  MSG_FPAIFAIL         0x7f2e7873
#define  MSG_FPAGFAIL         0x7f2ec073
#define  MSG_OUTNTFND         0x7f21b0fb
#define  MSG_PKT2SMAL         0x7f3493a3
#define  MSG_PKT2FEW          0x7f0d2c6b
#define  MSG_ALOCFAIL         0x7f143873
#define  MSG_TNM2LONG         0x7f094bbb
#define  MSG_GAPBEGIN         0x7f310e37
#define  MSG_GAPEND           0x7f0308bb

#endif    /* CMX_DOXYGEN */

#endif    /* MSG_MSGS_H */
```

Figure 3 The file `MSG_msgs.h`

1.1 Naming Convention

Putting this package together, I frequently lost track of which header files contained `msg2src` processor output and which were real. To deconfuse myself I invented the following convention. Postulating a `.msg` file defining messages for a facility called BAR in a package called FOO, then the `.msg` file is named `BAR_msgs.msg`. The corresponding `msg2src` output files are `BAR_msgs.h` and `BAR_msgs.c`. This looks very silly for the MSG package itself where I end up with a `.msg` file called `MSG_msgs.msg`, but for a consumer of this FOO/BAR combination, including the message header file would look like:

```
.
.
#include "FOO/BAR_msgs.h"
.
.
```

This immediately identifies:

- The package providing the facility (FOO).
- The facility name (BAR), which is also the prefix on all of this facility's message mnemonics.
- Self identifies the included file as one of these message files.

Try it … I think you'll like it!

1.2 Facility Numbers

All facilities must be given a facility number in the range 0-255. No two facilities may have the same facility number (or name for that matter). Problem: how does the user know how to pick a unique facility number?

I am proposing an administrative solution. The facility number space will be broken into 16 blocks of 16 numbers. Blocks will be assigned to individual developers who are free to assign facility numbers from their own block. The initial allocation is as follows:

Block	Range (decimal)	Range (hex)	Assigned to
0	0-15	0x00-0x0f	Tony Waite
1	16-31	0x10-0x1f	JJ Russell
2	32-47	0x20-0x2f	Curt Brune
3	48-63	0x30-0x3f	Sergio Maldonado
4	64-79	0x40-0x4f	James Swain
5	80-95	0x50-0x5f	Michael Monirzad
6	96-111	0x60-0x6f	Kim Lo
7	112-127	0x70-0x7f	Dan Wood
8	128-143	0x80-0x8f	Ray Caperoon
9	144-159	0x90-0x9f	Reserved
10	160-175	0xa0-0xaf	Reserved
11	176-191	0xb0-0xbf	Reserved
12	192-207	0xc0-0xcf	Reserved
13	208-223	0xd0-0xdf	Reserved
14	224-239	0xe0-0xef	Reserved
15	240-255	0xf0-0xff	Tony Waite

Table 1 Facility number block assignments

1.2.0 Facility Number Usage In Block 0

These are the assignments within block 0:

Facility	Name	Comment
0	(none)	Never assigned. Facility 0 is used to grandfather Unix error codes.
1		Unassigned
2		Unassigned
3		Unassigned
4		Unassigned
5		Unassigned
6		Unassigned
7		Unassigned
8		Unassigned
9		Unassigned
10		Unassigned

Facility	Name	Comment
11		Unassigned
12		Unassigned
13		Unassigned
14		Unassigned
15		Unassigned

Table 2 Usage of facility numbers 0-15 (block 0)

1.2.1 Facility Number Usage In Block 15

These are the assignments within block 15:

Facility	Name	Comment
240		Unassigned
241		Unassigned
242		Unassigned
243		Unassigned
244		Unassigned
245		Unassigned
246		Unassigned
247		Unassigned
248		Unassigned
249		Unassigned
250		Unassigned
251		Unassigned
252	MTS	Messages used in the Message Test Suite.
253	M2S	Messages generated by the `msg2src` executable.
254	MSG	The message system itself.
255	(none)	Never assigned. Facility 255 is used to grandfather Unix error codes.

Table 3 Usage of facility numbers 240-255 (block 15)

2 Quick User Walkthrough

This section is intended to walk through some trivial coding examples to show how a developer would incorporate MSG into an application. A later section will deal with how a developer incorporates MSG into the package build process.

2.0 Sending A Message

The call to output a message is very analogous to `printf`. Here is a shard of code notionally from the message package itself to illustrate:

```
0  #include “MSG/MSG_msgs.h”

1  unsigned int my_routine
   (
       unsigned int       size,
       char           **buffer
   )
   {

       unsigned int
           status;

       static char
2         *rtn = “my_routine”;
       .
       .
       .
       if( (*buffer = malloc( size )) == NULL )
       {
           status = MSG_ALOCFAIL;
3          MSG_signal( &status, rtn, 1, size );
4          return( status );
       }
       .
       .
       .
5      return( MSG_SUCCESS );
   }
```

Figure 4 A trivial illustration of calling `MSG_signal()`

Notes:

0. This is how MSG exports its list of messages (in computer digestible form).
1. When you get used to this system, it's amazing how almost every routine turns into an `unsigned int` so that the code that generates the message can also be propagated back to the caller. Note the return statements in this routine.
2. The signaled message captures the routine name as part of the message, so please provide (an accurate) one.
3. The heart of the system. The first three arguments are mandatory:
    - A message code *passed by reference*. This looks a little strange at first sight, but `MSG_signal()` is going to set the top bit of this code if it's successful. It does this so that as a message code is passed back through stack frames, higher stack frames trying to resignal the same message will be unsuccessful.

Warning: the fact that `MSG_signal()` can (and usually does) change the value of `status` can make testing the values of message codes hazardous. Please pay special attention in the next section on how to receive and deal with message codes.

    - The routine name.
    - The number of parameters that follow. This will be dealt with in more detail in section 2.5.
    - Subsequent arguments are variable in the same way that arguments to `printf` are variable. If you look back at the definition of the code `MSG_ALOCFAIL`, you will see that it's expecting a single integer argument.
4. As a courtesy to the caller of this routine, it's better to return `status` than the code `MSG_ALOCFAIL`. The variable `status` has already had its "I've been signaled" flag set.
5. `MSG_SUCCESS` is not signaled, so it can be returned raw.

2.1 Receiving A Message Code

Now let's assume that I'm the caller of `my_routine`. How do I deal with the message code? I will start with a code shard that contains the most common pratfall and then go through the process of improving the code:

```
#include “MSG/MSG_msgs.h”

unsigned int my_caller()
{
    unsigned int;
        status;

    char
       *buffer;

    static char
       *rtn = “my_caller”;

    .
    .
    .
    status = my_routine( 1000, &buffer );
    if( status == MSG_ALOCFAIL )
    {
        return( status );
    }
    .
    .
    .
    return( MSG_SUCCESS );
}
```

Figure 5 How not to deal with a message code

This looks very plausible but there is an outright bug in this version. The value of `status` returned from `my_routine()` has its uppermost bit set to indicate that it’s already been signaled. The value will *not* be equal to `MSG_ALOCFAIL`.

I realize that I’m harping on about this “message has been signaled” bit, but this is the most common error I’ve seen using a system like this.

MSG provides some relief in the form of three preprocessor macros (and before someone points it out: Yes it is unconventional to use lower case macros, but you’re going to be seeing a lot of these and I find the lower case versions much easier to read accurately!):

Macro	Definition	Meaning
`_msg_success(x)`	`(((x) & 1) == 0)`	True if the severity of message x is either success or information
`_msg_failure(x)`	`(((x) & 1) != 0)`	True if the severity of message x is either warning or error
`_msg_match(x,y)`	`(((x) & MSG_M_MATCH) == ((y) & MSG_M_MATCH))`	True if messages x and y match (ignoring the “I’ve been signaled” bits).

Table 4 MSG preprocessor macros to help interpret message codes

Rewriting the above code shard…

```
#include “MSG/MSG_msgs.h”

unsigned int my_caller()
{
    unsigned int;
        status;

    char
       *buffer;

    static char
       *rtn = “my_caller”;

    .
    .
    .
    status = my_routine( 1000, &buffer );
    if( _msg_failure( status ) )
    {
        return( status );
    }
    .
    .
    .
    return( MSG_SUCCESS );
}
```

Figure 6 A better way to receive a message code

This has at least removed the bug.

The test could also have been written `if( _msg_match( status, MSG_ALOCFAIL ) )`. The `_msg_match` method is very precise but can get cumbersome in the case of a typical routine that returns just one code to indicate success, but then layers on all sorts of error severity codes to indicate all the different ways it can fail. Common practice is therefore to use the `_msg_success` or `_msg_failure` macros unless the caller has a special need to sort through the precise details of what went wrong in the called routine.

This code is certainly better but could still be improved. Starting with an example that signals a `malloc()` failure was not an idle choice. `malloc()` is a very low level routine and has no context to report anything more than its own failure. As the call stack unwinds, greater and greater context becomes available. In the above example the routine `my_caller()` probably knows the purpose of the memory it requested `my_routine()` to allocate. In the event of failure, why not signal this too:

```
#include “MSG/MSG_msgs.h”

unsigned int my_caller()
{
    unsigned int;
        status;

    char
       *buffer;

    static char
       *rtn = “my_caller”;

    .
    .
    .
    status = my_routine( 1000, &buffer );
    if( _msg_failure( status ) )
    {
        status = MSG_MOREFAIL;            /* I just made this up …          */
        MSG_signal( &status, rtn, 1,   ); /* …so I don’t know the arguments */
        return( status );
    }
    .
    .
    .
    return( MSG_SUCCESS );
}
```

Figure 7 Using `MSG_signal()` to generate a call stack trace

This is very close to generating a full call stack trace. MSG can get very noisy when used this way and it might not be appropriate in all cases, but it can be a very useful technique.

2.2 Signaling Levels

A system like MSG always has the implicit flavour of an *error* signaling system. By default MSG does behave this way, but it also provides facilities to modify that behaviour.

Without intervention, the `MSG_signal()` routine will simply return if the message severity is not `E` (error). That’s not always what the user wants. Take the example of a complicated initialization routine that at the end of processing would like to announce that initialization completed successfully. The following code simply won’t work:

```
#include “MSG/MSG_msgs.h”

unsigned int my_initialization()
{
    unsigned int;
        status;

    static char
       *rtn = “my_initialization”;

    .
    .
    /* Lots of very complicated initialization */.
    .
    .
    status = MSG_INITGOOD;                      /* Severity level S        */
    MSG_signal( &status, rtn, 0 );              /* Ignored by MSG_signal() */
    return( status );
}
```

Figure 8 Unsuccessful attempt to signal a success message

A temporary redefinition of the MSG default signaling level can remedy the situation:

```
#include “MSG/MSG_msgs.h”
#include “MSG/MSG_pubdefs.h”

unsigned int my_initialization()
{
    MSG_Level
        level;

    unsigned int
        status;

    static char
       *rtn = “my_initialization”;

    .
    .
    /* Lots of very complicated initialization */.
    .
    .
    status = MSG_INITGOOD;                    /* Severity level S           */
    level = MSG_setLevel( MSG_LVL_SUCCESS );
    MSG_signal( &status, rtn, 0 );            /* Not ignored by MSG_signal() */
    level = MSG_setLevel( level           );
    return( status );
}
```

Figure 9 Successful attempt to signal a success message

`MSG_setLevel()` returns the current signaling level as part of the setting of a new level. This is convenient a couple of lines later when the old signaling level needs to be restored. `MSG_signal()` will signal codes whose severity is *greater than or equal to* the currently set signaling level. The possible signaling levels are:

- `MSG_LEVEL_SUCCESS`

- MSG_LVL_INFORMATION
- MSG_LVL_WARNING
- MSG_LVL_ERROR

Note that with this definition, it's not in fact possible to suppress codes with severity MSG_LVL_ERROR.

2.3 Trace Buffers

Analogous to the level setting system is the trace buffer system. Every message generated by MSG_signal() captures the number of a trace buffer. By default this is just zero, but the number is under user control. Suppose all initialization completion messages were conventionally annotated as trace buffer 42:

```
#include "MSG/MSG_msgs.h"
#include "MSG/MSG_pubdefs.h"

unsigned int my_initialization()
{
    MSG_Level
        level;

    unsigned int
        status;

    unsigned short
        trace;

    static char
       *rtn = "my_initialization";

    .
    .
    /* Lots of very complicated initialization */.
    .
    .
    status = MSG_INITGOOD;
    level = MSG_setLevel( MSG_LVL_SUCCESS );
    trace = MSG_setTrace( 42              );
    MSG_signal( &status, rtn, 0 );
    trace = MSG_setTrace( trace           );
    level = MSG_setLevel( level           );
    return( status );
}
```

Figure 10 Attaching a trace buffer number to a message

2.4 Signaling Level And Trace Buffer Number Scope

This discussion will anticipate the introduction of multi-threaded operation, but belongs here to associate it with the description of signaling levels and trace buffers numbers.

Signaling levels and trace buffer numbers are maintained *on a per thread/task basis*. That means that if task A sets the signaling level down to information, but gets interrupted by task B before it

can signal, task B does *not* inherit the signaling level from task A. Similarly for trace buffer numbers.

2.5 Signaling With Partial Formatting

The third argument to `MSG_signal()` is a count of the number of parameters following. In the standard case, this number should be the same as the number of tokens expected by the message formatting string defined in the `.msg` file (and the parameters that follow would normally agree in both count and type with the message formatting string). What should the developer do if the information to fill in the parameters in not available? This situation most commonly arises when a called routine reports a warning level code. With the default signaling level, even if the called routine in turn calls `MSG_signal()`, the message is ignored ("warning" is less than "error"). The calling routine has enough context to know that the warning is really an error and would like to signal it. The first instinct is to use the `MSG_setLevel()` routine to drop the signaling level to warning and then signal the message. It's not until the writer reaches the next line of code that problems set in … the message needs parameters to put into the message string, but the calling routine doesn't know what they should be! Only the called routine knew them!

Messages can be signaled with *partial* formatting. That is the real purpose of the third argument to `MSG_signal()`. The third argument should always accurately reflect the number of parameters that follow, even if the number of parameters does not agree with the number of parameters that the formatting string implies. The only restriction is that any parameters that do follow must agree in type and order with the substitutions implied by the formatting string. In the example given above, if the calling routine doesn't know how to fill in any of the parameters (a common case), it is perfectly legal to code the third argument to `MSG_signal()` as zero, and omit all further parameters.

If parameters are omitted, the output formatting simply copies the conversion string to the output string, thus the output string may contain something like `0x%08x` instead of `0x12345678`.

2.6 Summary

This section has illustrated almost everything the average user needs to know to make the message system work at run time. The number of calls available is really very small:

- `MSG_getLevel()`
- `MSG_getTrace()`
- `MSG_setLevel()`
- `MSG_setTrace()`
- `MSG_signal()`

And the number of macros is even smaller:

- `_msg_success`
- `_msg_failure`
- `_msg_match`

Obviously, there are a number of other calls needed to set up and shut down the message system, but these are rarely encountered by the average user. Section 4 goes into more detail of what the message system is really doing at run time and how to start and stop it.

3 MSG At Build Time

3.0 The Requirements File

CMX/CMT provides direct support for processing `.msg` files. It comes in the form of a new document type called (you guessed) `msg`.

This is a notional shard of a requirements file that processes a `.msg` file and uses the output files in a subsequent constituent:

```
0  # ----------------------------------------------------------------------------
   # Invoke CMX functionality
   # ----------------------------------------------------------------------------
   use CMX
   private

   # ----------------------------------------------------------------------------
   # Process BAR_msgs.msg to produce BAR_msgs.c and BAR_msgs.h
   # ----------------------------------------------------------------------------
1  document msg bar_msgs \
       BAR_msgs.msg

   # ----------------------------------------------------------------------------
   # Shareable: foo_shr
   # ----------------------------------------------------------------------------
   macro foo_shr__buildtags "sun-gcc linux-gcc mv2303 mv2304 mcp750 rad750"
   macro foo_shr__linkshr   "" \
       linux-gcc      "$(PBS_pbs__shr) $(CMX_cmx_asBuilt__shr) -lpthread -lc" \
       sun-gcc        "$(PBS_pbs__shr) $(CMX_cmx_asBuilt__shr) -lc" \
       mv2303         "$(PBS_pbs__shr) $(VXW_vxw_flight__shr)" \
       mv2304         "$(PBS_pbs__shr) $(VXW_vxw_flight__shr)" \
       mcp750         "$(PBS_pbs__shr) $(VXW_vxw_flight__shr)" \
       rad750         "$(PBS_pbs__shr) $(VXW_vxw_flight__shr)"

   document shr foo_shr \
       FOO_this.c \
       FOO_that.c \
2      BAR_msgs.c \
       FOO_the_other.c
```

Figure 11 Requirements file shard

Notes:

0. I started at this line just to orient people in the requirements file.
1. This is the new document type. It only accepts files with the extension `.msg`. More than one file can be defined. The corresponding `.c` file is always placed in the same directory as the `.msg` file (so this mechanism can be used in `/ptd` directories as well) and the `.h` file is always placed in the package's export directory. The constituent name has very little significance and is not the stem for a bunch of magic macros (document type `msg` does not support any).
2. The `.c` output file from the "message compilation" step is used in a regular constituent. The only trick here is that the constituent that produces the `.c` file (the `msg` type constituent) must appear in the requirements file *before* the constituent that uses it (in this case the `shr` type constituent). It is one of the lesser known facts about CMT make processing that constituents are built in the order in which they appear in the requirements file (or in exactly the reverse order if it's a `make clean`).

3.1 Message Files And CVS

Note that despite the fact that the output files from the `msg2src` step appear in the source tree of a package, they are really *derived* products. Thus in the example given, the files `BAR_msgs.h` and `BAR_msgs.c` should *not* be added to the CVS repository. The `.msg` files *should* be added to the CVS repository.

For those who delight in nit-picking, I do realize that I myself disobeyed this rule in the MSG package. This was done to break the bootstrap cycle (MSG uses `.msg` files to produce messages, but it can't do the `msg2src` step because the MSG package provides the `msg2src` executable as a product of the build). The MSG package has to do hand-held builds of all its `.msg` files (and you should be grateful that you don't have to mess with this stuff!)

Dan Wood has invented a fine method of annotating `msg2src` output files as derived files and for keeping the CVS repository clean. He has put a `.cvsignore` file in each of his package's `/src` and `/<pkg>` directories in which he lists the derived files. See package `ZLIB` (`V1-0-0` or later) for an example.

4 MSG At Run Time

This section takes a peek behind the curtain at what the message system is really doing at run time and how to start and stop the message system. To start the ball rolling, the first item will look at message *disposition*. The sharp eyed may have noticed that to date, there has been no description of what happens to a message generated by a call to `MSG_signal()`. Curiously enough the technical answer is … nothing! The message system must be configured when it is started with one or more…

4.0 Output Processors

Output processors can be attached and detached to the message system by making calls to the routines `MSG_attachOutputRtn()` and `MSG_detachOutputRtn()`. These calls will only be honored when the message system is in state `INITIALIZED` (see the state diagram in section 4.1.0.0). A separate constituent in the MSG packet (`msg_print`) implements one such processor, a simple print routine with a rudimentary capacity to tailor what's printed.

Each output processor is called for each message generated by a call to `MSG_signal()`. Each processor is also given one opportunity to initialize itself (the `MSG_startTask()` routine calls all output processors with an initialization flag) and one opportunity to clean up (the `MSG_stopTask()` routine calls all output processors with an exit flag).

4.1 Multi-Threaded And Single-Threaded Operation

The message system comes in two flavours: A multi-threaded version and a single-threaded version. The multi-threaded version is available in host Unix and embedded VxWorks environments The single threaded version is only available in host Unix environments. The single-threaded version is in general simpler, so most of this section will concentrate on the multi-threaded version. A small piece at the end will describe places where the single-threaded version differs.

4.1.0 Multi-Threaded Operation

Be aware that in multi-threaded mode, MSG uses a number of facilities in PBS and that PBS must be initialized (`PBS_initialize()`) before MSG can operate.

4.1.0.0 Starting And Stopping

The message system does require starting/configuration and (optionally) stopping. The state diagram and associated calls for starting and stopping looks like:

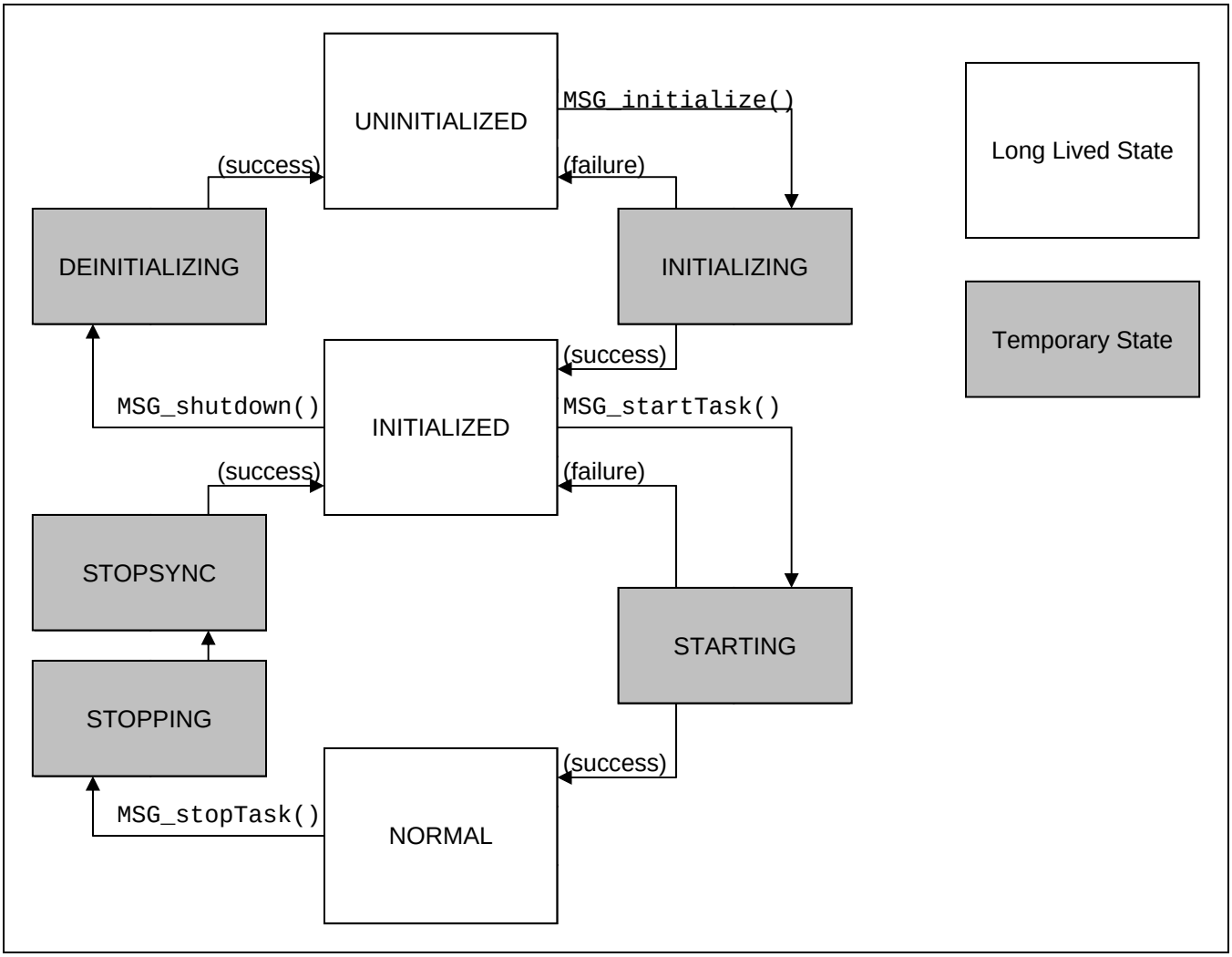

Figure 12 Starting and stopping the message system.

4.1.0.0.1 MSG_initialize()

The initialization call allocates all resources needed by the message system. These include:

- The memory for the message packets along with the Fixed Packet Allocator (FPA) control block.
- The memory for a fork control block.
- A read-write interlock control block.

The call takes a pointer to a parameter block describing the packet allocation. Storage for the parameter block need not be made persistent. The two members of the parameter block are:

- `pkt_len`: Length of a message packet. Message packets must always be a minimum length (which I can't fix yet until I get some idea of how long strings are going to be!). A sub-minimum packet length will be rejected. Sensible packet lengths are in the range 192

to 256 bytes (I've been using 256). The packet length must also be a multiple of eight to keep a `long long` variable in the packet aligned in all packets. Processing in `MSG_initialize()` will always round `pkt_len` to the next highest multiple of eight.

- `pkt_cnt`: Count of message packets. The bare minimum is three (see section 4.1.0.2 for the reason), but this would be a very uncomfortable way to run. A more sensible number is of order of twice the number of tasks in the system (so that they can all be chewing on a packet at the same time and still have some left over). I've been using 32 packets.

With an allocation of 32 packets at 256 bytes per packet, the message system is running on about 8 kByte of memory. Not exactly a hog.

4.1.0.0.2 State INITIALIZED

This is the only state in which output processors can be attached and detached.

4.1.0.0.3 MSG_startTask()

`MSG_startTask()` performs two distinct functions:

- It spawns the message task (using the PBS facility FORK).
- It cans through all attached output processors making an initialization call to each of them.

Like `MSG_initialize()`, `MSG_startTask()` takes a parameter block which it uses to set up the task. Storage for the parameter block need not be made persistent. The parameter block is the standard TASK attributes block. Full documentation of the TASK attributes block can be found in the PBS documentation. The following examples demonstrate some of the techniques available to tailor the message task with the task attributes block:

The task attributes block cannot be used to manipulate the task *name*. That's MSG's property!

```
.
.
#include “PBS/TASK.h”
.
.

unsigned int my_initialization()
{
    .
    .
    TASK_attr
        attr;
    .
    .

    /*
     | Initialize the task attributes block accepting all defaults.
    */
    TASK_attr_init( &attr );
    .
    .
}
```

Figure 13 Task attributes block, accepting all defaults

```
.
.
#include “PBS/TASK.h”
.
.

unsigned int my_initialization()
{
    .
    .
    TASK_attr
        attr;
    .
    .

    /*
     | Initialize the task attributes block, defining the size of the stack
     | but leaving it to the TASK routines to allocate it.
    */
    TASK_attr_init( &attr );
    attr.stack_size = 0x2000;
    .
    .
}
```

Figure 14 Task attributes block, managing the size of the stack

```
.
.
#include “PBS/TASK.h”
.
.

unsigned int my_initialization()
{
    .
    .
    TASK_attr
        attr;

    unsigned int
        size = 0x2000;

    char
       *addr;
    .
    .

    /*
     | Initialize the task attributes block, managing both the size and
     | allocation of the stack.
    */
    addr = (char *) MBA_alloc( size );
    TASK_attr_init( &attr );
    attr.stack_addr = addr;
    attr.stack_size = size;
    .
    .
}
```

Figure 15 Task attributes block, managing both the size and allocation of the stack.

```
.
.
#include “PBS/TASK.h”
.
.

unsigned int my_initialization()
{
    .
    .
    TASK_attr
        attr;
    .
    .

    /*
     | Initialize the task attributes block, specifying the task priority.
     | The following example says “increase the message task’s priority four
     | levels relative to this calling task”.
    */
    TASK_attr_init( &attr );
    attr.priority = TASK_priority_number_compute
                    ( 4, TASK_ K_PRIORITY_TYPE_REL, NULL );;
    .
    .
}
```

Figure 16 Task attributes block, managing the task priority.

4.1.0.0.4 State **NORMAL**

The only notable feature of the `NORMAL` state is that it is the only state in which `MSG_signal()` will process calls.

4.1.0.0.5 MSG_stopTask()

`MSG_stopTask()` stops message processing. Easy to say, less easy to do in a pipelined system. To stop the message system, `MSG_stopTask()`:

- Immediately puts the message system into state `STOPPING` so that any further calls to `MSG_signal()` are ignored.
- Puts a special message on the message task processing queue.
- “Joins” the message task so that the calling task is stalled until the message task exits.

On the message task side:

- Packet processing continues normally until the special message is seen.
- The message system is put into state `STOPSYNC`.
- The message task continues processing, but checks after each message to see if all packets are now either on the packet free list or are specially allocated to the message system itself. Once this is true, the message task exits (which satisfies the “join” and the task calling `MSG_stopTask()` is unblocked).

Back on the `MSG_stopTask()` side:

- Once unblocked, `MSG_stopTask()` scans through all attached output processors making an exit call to each.
- Puts the message system into state `INITIALIZED` and exits.

4.1.0.0.6 MSG_shutdown()

`MSG_shutdown()` deallocates all resources associated with the message system (mostly memory).

The deallocation is very thorough. It also deallocates all information about output processors.

4.1.0.1 Steady State Operation

The message system maintains a block of message packets which get cycled through tasks wanting to signal messages. The life cycle of a message packet is depicted in the following figure:

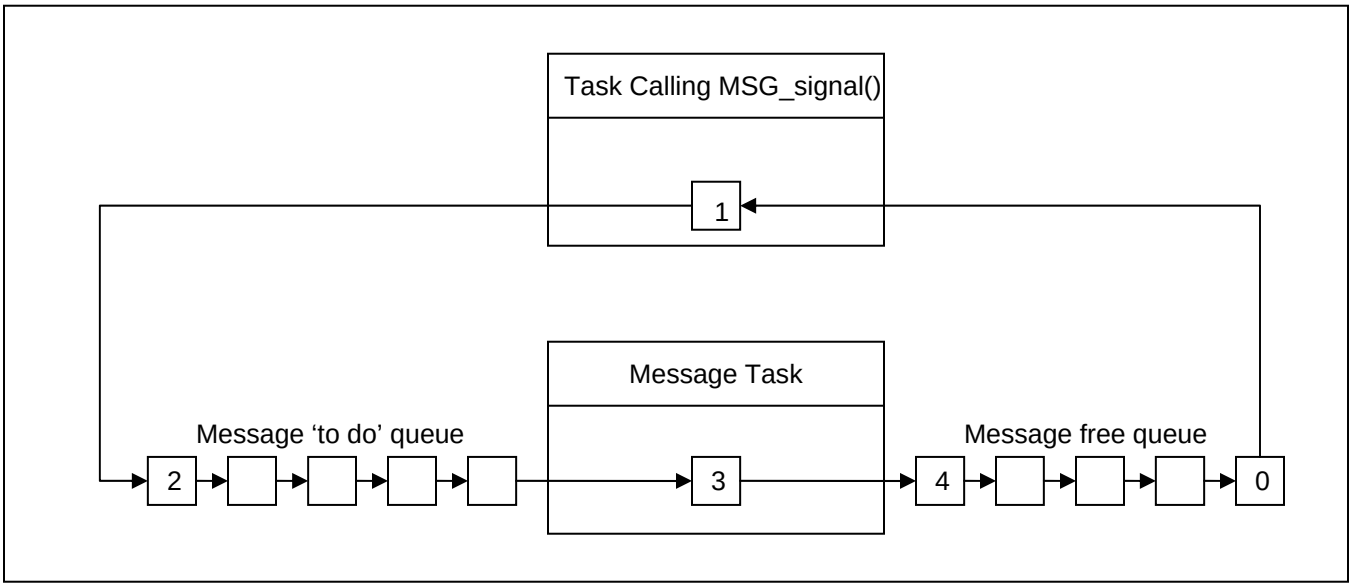

Figure 17 Life cycle of a message packet

0. When a task calls `MSG_signal()`, `MSG_signal()` allocates a message packet from the message free queue.
1. `MSG_signal()` captures information, including the variable parameter list, into the packet.
2. `MSG_signal()` places the packet on the processing queue of the message task.
3. When it works its way to the front, the message task picks the packet up, does the real message formatting, then runs it through all attached output processors.
4. Once the message processors are complete, the message task puts the packet back on the free list.

4.1.0.2 When Resources Run Out

The message system runs on a finite number of message packets. Under unusual conditions (many tasks trying to signal simultaneously or the message task being starved of cycles while its input queue is long), the system can run out of buffers.

Message copes with this situation by pre-allocating a pair of lookaside packets. When a task trying to signal can't allocate a packet from the free list, it will instead use the first of the lookaside packets to signal a "blackout begin" message. All messages signaled during blackout are simply dropped on the floor. When the message output task recognizes that it has freed sufficient packets to end the blackout, it will use the second lookaside packet to signal a "blackout end" message (this message includes a count of dropped messages) and re-enables signaling.

Through the wonders of multi-threaded environments it is in fact possible for real messages to appear on the message output queue between the "begin blackout" and "end blackout" messages. This occurs when task A allocates (successfully) a signaling packet, but before queueing the message, is interrupted by task B. Task B's packet allocation fails and task B queues the begin blackout message. When task B gives up execution, task A can be re-scheduled at which point task A completes its attempt to put out its message.

4.1.0.3 Interrupt Level Operation

This information only applies to the embedded systems. While PBS facilities give the illusion that application code running multi-threaded on Unix hosts is receiving interrupts (from timers and whatnot), the signals are generated by another thread under JJ's control. To MSG this thread looks like any other thread and is treated the same way.

The message system is interrupt safe. Calling `MSG_signal()` from interrupt level will not cause the system to die.

It's another question whether it is wise to call `MSG_signal()` from interrupt level. The cost is increased interrupt latency if `MSG_signal()` runs slowly. Tests on one of our `mcp750` boards (a PPC 750 running at 233 MHz) returned execution times of order 4 μsec which is acceptable for interrupt latency, but I would not place too much faith in this number as it was generated in a test running a tight loop where both instructions and data were probably stored in cache memory. I think 5-10 μsec would probably be a better estimate.

Bottom line, call `MSG_signal()` from interrupt level if you must, but don't make a habit of it!

Notionally, any code executed at interrupt level runs outside the context of a task. Thus the concept of task specific reporting levels and trace buffers is meaningless. Fortunately interrupt handling is strictly serialized so MSG can (and does) keep an extra copy of the reporting level and trace buffer number specific to interrupt level. Calls to routines like `MSG_setLevel()` or `MSG_setTrace()`, if called from interrupt level, will modify the interrupt level reporting level and trace buffer respectively. The same calls from task level modify the task specific versions.

Please take a moment to think through the implications of this implementation. If interrupt level code changes the reporting level and does not restore it to its original value, the new value will persist into the next piece of interrupt code that executes. This will surprise and confuse. Any changes to the reporting level or trace buffer number at interrupt level should be restored before

exiting the interrupt.

4.1.1 Single-Threaded Differences

Single-threaded operation differs from multi-threaded in the following ways:

- There is no need to call `PBS_initialize()` (though it does no harm). Technically speaking the single-threaded version of MSG *does* use PBS facilities, but not any that are set up by the PBS initialization call.
- The parameter block to `MSG_initialize()` is still required. In the single-threaded case, the minimum packet allocation (three) is also a sensible way to run. On the other hand, over-allocating packets (for symmetry with a multi-threaded implementation for instance), does no harm other than waste a little memory.
- There is no spawned message task. The call to `MSG_startTask()` is still required (to run the initialization phase of the output processors), but the parameter block provided in the `MSG_startTask()`call is ignored.
- All calls to `MSG_signal()` are processed synchronously.
- The shutdown is synchronous and does not require (or implement) any form of “draining” logic.
- Single-threaded operation has no concept of tasks/threads and consequently no concept of names for them. To compensate, it is possible to set an arbitrary name using the `MSG_setTask()` routine. The name will be reported as the “task name” for all messages.

4.2 Summary

A typical message start up and shut down would look something like (this would work both multi-threaded and single-threaded):

```
#include “PBS/PBS.h”
#include “PBS/TASK.h”
#include “MSG/MSG_pubdefs.h”
#include “MSG/MSG_printProc.h”

unsigned int my_initialization()
{
    TASK_attr
        attr;

    MSG_InitPrm
        init;

    MSG_OutputRtn
       *prt;

    unsigned int
        status;

    /*
     | If it hasn’t been done elsewhere, initialize PBS.
    */
    PBS_initialize( 0, 0 );

    /*
     | Initialize the message system.
    */
    init.pkt_cnt =  32;
    init.pkt_len = 256;
    status = MSG_initialize( &initPrm );
    if( _msg_failure( status ) )
    {
        return( status );
    }

    /*
     | Attach a simple printing output processor (and discard the handle
     | returned by MSG_attachOutputRtn … I’m not planning on using the
     | handle to “MSG_detachOutputRtn”).
    */
    status = MSG_attachOutputRtn( &prt, MSG_print, NULL );
    if( _msg_failure( status ) )
    {
        return( status );
    }

    /*
     | Initialize the task attributes block.  This version simply accepts all
     | defaults.  For more extensive examples of initializing a task attributes
     | block, see section 4.1.0.0.3.
    */
    TASK_attr_init( &attr );

    /*
     | Start the message task.
    */
```

```
        status = MSG_startTask( &attr );

        /*
         | That's all folks!
        */
        return( status );
    }
```

Figure 18 Initializing/starting the message system.

```
    #include "MSG/MSG_pubdefs.h"

    unsigned int my_shutdown()
    {
        unsigned int
            status;

        /*
         | Stop the message system.
        */
        status = MSG_stopTask();
        if( _msg_failure( status ) )
        {
            return( status );
        }

        /*
         | Shut down the message system.
        */
        status = MSG_shutdown();

        /*
         | That's all folks!
        */
        return( status );
    }
```

Figure 19 Stopping/shutting down the message system.

5 Tips And Tricks

5.0 What Severity Code And When To Signal

There is a lot of scooch room in how a system like MSG can be run. Adopt the rules:

- Signal *all* messages of any severity.
- Reduce the signaling level to `MSG_LVL_INFORMATION` globally.

and the system becomes a running log of activity … and very, very noisy! Adopt the rules:

- Only signal error severity messages.
- Keep the signaling level at `MSG_LVL_ERROR`.

and the system turns into a pure error log.

Which leads inevitably to the question, what is an error?

I have never been able to answer this one satisfactorily. The situation so often arises that a piece of code can identify that something out of the ordinary is happening, but because of reduced context, is not in a position to judge whether this is an error or not. Hence the frequent use of language like “condition code” as opposed to “error code”.

For the moment I would like to propose the following guidelines:

- Keep the default signaling level at `MSG_LVL_ERROR`.
- Deciding whether a “this can’t be good” condition is an `ERROR` or a `WARNING` is up to the developer. This can be a tough decision, but don’t waste too much effort on it. The severity can be changed later (granted this requires a rebuild of packages that use the message code, but only rarely requires recoding).
- Signal any `ERROR` or `WARNING` severity messages immediately upon detection. Signaling `SUCCESS` or `INFORMATION` severity messages is certainly allowed but not common.
- For special `SUCCESS` or `INFORMATION` severity messages that should be signaled, drop the signaling level temporarily and locally.

5.1 The Minimalist `.msg` File

At some point every developer discovers the following trick message file:

```
# ------------------------------------------------------------------------
# CVS $Id$
# ------------------------------------------------------------------------

--FACILITY FOO 42

GENERICS S "%s"
GENERICI I "%s"
GENERICW W "%s"
GENERICE E "%s"
```

Figure 20 The minimalist `.msg` file

The developer then uses `sprintf` to prepare text strings to be signaled.

Please don’t do it! Remember that while text strings are very digestible by humans, computers prefer numbers and what you are returning to the callers of your routines is a very limited set of numbers.

The one time I find this trick useful is during early development of a new package. I will indeed write this as my first pass `.msg` file and use (usually literal) strings in messages. Once the package has settled down though and the required set of messages stabilized, I go back through and retrofit a more reasonable (and usually more informative) set of messages.

6 Future Development

MSG is not complete. I have already down-scoped it twice to try to get it out there and get it into use. Herewith a few ideas for extending it.

6.0 Output Processors

MSG only provides a single output processor to print messages to `sysout`. This is not tremendously useful on a satellite!

Obviously, more output processors will have to be written. I fully expect to see (and very probably write) an output processor to push messages across 1553. Not quite so obvious is that this output processor will very likely *not* be part of the MSG package (the MSG package should as far as possible not depend on other packages because most packages will want to depend on the MSG package … this is why MSG output processing is constructed as plug-in callbacks!)

6.1 Common Run Time Improvements

6.1.0 CPU Identification

`MSG_signal()` is supposed to capture the identity of the CPU sending the message. There is no implementation behind this yet.

6.2 Single-Threaded Specific Run Time Improvements

6.3 Multi-Threaded Specific Run Time Improvements

6.4 Ground Based Additions

6.4.0 A Logging Database In The IOC

Presuming that MSG will one day drive messages either across 1553 or across the science data interface (or both), it would be useful to interpret the messages on the ground and insert them

into a random access database. This was the intent behind capturing a lot of “key” information with the message. This is based on the experience we had with the SLDERR0R system. Messages can form a useful historical log of events (particularly anomalous ones). It can be very informative to be able to ask a database “Show me all the messages from task <task> on cpu <cpu> between time <time_lo> and time <time-hi>”.

6.4.1 A Message Database In The MOC

This one is extremely vague. It was only recently that I discovered that the MOC would not be able to interpret variable length telemetry (i.e. strings). If MOC had access to the message definitions in a database on the ground then just the message code (no strings) could be used to look up at least the formatting string and display something on, for instance a scrolling screen.