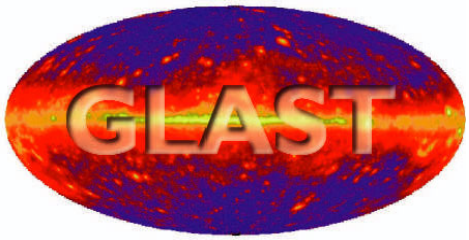

LAT Flight Software

MSG Manual

Type:	User Manual
Version:	V1-2-0
Author:	A.P.Waite
Created:	27 October 2003
Updated:	12 May 2004
Printed:	12 May 2004

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_report()` macro (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 place the processed contents of the `.msg` file in a look-up table 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 `M2S_msgs.msg` followed by its derived files:

Yes indeed the `msg2src` 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: M2S_msgs.msg,v 1.2 2003/10/31 04:28:30 apw Exp $
   #
   # Messages associated with the msg2src executable.
   # ------------------------------------------------------------------------
1
2  --FACILITY M2S 253

   # ------------------------------------------------------------------------
   # The canonocal success message
   # ------------------------------------------------------------------------
3  SUCCESS  S "Success"

   # ------------------------------------------------------------------------
   # Command line errors
   # ------------------------------------------------------------------------
   BADNPARM E "Too %s parameters to %s command"
   BADFLFMT E "%s is not a .msg file"
   BADFLLEN E "%s too short to be the name of a .msg file"
   BADFLNAM E "%s does not end with file extension .msg"
   BADQUAL  E "Cannot parse qualifier \"%s\""
   QULNEGAT E "Qualifier --%s is not negatable"
   QULVALUD E "Qualifier --%s never takes a value"
   QULNOVAL E "Qualifier --%s always takes a value"
   QULUNDEF E "Qualifier %s not recognized"

   # ------------------------------------------------------------------------
   # File manipulation errors
   # ------------------------------------------------------------------------
   FILOPEN  E "Error opening file %s"
   FILWRITE E "Error writing file %s"
   FILCLOSE E "Error closing file %s"

   # ------------------------------------------------------------------------
   # Internal errors
   # ------------------------------------------------------------------------
   ALOCFAIL E "Cannot allocate %d bytes for %s"
   BADTOKEN E "Cannot parse token \"%s\""
   DUPMSGID E "Duplicate message IDs for names %s and %s"
   DUPMSGNM E "Duplicate message name %s"
```

Figure 1 The file `M2S_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!), but there are constraints:
     - Special escape characters (newline, tab, bell, …) are forbidden. This is detected by the `msg2src` executable and will produce appropriate nastygrams.
     - The formatting string must not define more than sixteen parameters. This is detected by the `msg2src` executable and will produce appropriate nastygrams.

Warning: Very occasionally, `msg2src` will complain about ambiguous message mnemonics. This arises when two mnemonic names in the same `.msg` file encode to the same number. The only cure is to change message mnemonics until the ambiguity is removed. I've actually Monte-Carlo'd this effect and it usually requires 500 or more messages in a `.msg` file before there's any substantial probability of this happening.

Running `msg2src` on this `.msg` file produces the following `.c` and `.h` files. 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  M2S_msgs.c
    \brief Ctor/Dtor routines for message facility \b M2S (ID: \c 253, \c0xfd)
    \warning Machine generated code - NEVER edit by hand
    *//*-------------------------------------------------------------------------*/

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

    #ifdef __cplusplus
    extern "C" {
    #endif

    /*-------------------------------------------------------------------------*//*!
    \var   static const char MSG_StrList_M2S
    \brief String database for message facility M2S
    *//*-------------------------------------------------------------------------*/
    static const char MSG_StrList_M2S[649] =
    {
        0x53, 0x55, 0x43, 0x43, 0x45, 0x53, 0x53, 0x00,
        0x53, 0x75, 0x63, 0x63, 0x65, 0x73, 0x73, 0x00,
        0x46, 0x49, 0x4c, 0x4f, 0x50, 0x45, 0x4e, 0x00,
        0x45, 0x72, 0x72, 0x6f, 0x72, 0x20, 0x6f, 0x70,
        0x65, 0x6e, 0x69, 0x6e, 0x67, 0x20, 0x66, 0x69,
        0x6c, 0x65, 0x20, 0x25, 0x73, 0x00, 0x42, 0x41,
        0x44, 0x51, 0x55, 0x41, 0x4c, 0x00, 0x43, 0x61,
        0x6e, 0x6e, 0x6f, 0x74, 0x20, 0x70, 0x61, 0x72,
        0x73, 0x65, 0x20, 0x71, 0x75, 0x61, 0x6c, 0x69,
        0x66, 0x69, 0x65, 0x72, 0x20, 0x5c, 0x22, 0x25,
        0x73, 0x5c, 0x22, 0x00, 0x41, 0x4c, 0x4f, 0x43,
        0x46, 0x41, 0x49, 0x4c, 0x00, 0x43, 0x61, 0x6e,
        0x6e, 0x6f, 0x74, 0x20, 0x61, 0x6c, 0x6c, 0x6f,
        0x63, 0x61, 0x74, 0x65, 0x20, 0x25, 0x64, 0x20,
        0x62, 0x79, 0x74, 0x65, 0x73, 0x20, 0x66, 0x6f,
        0x72, 0x20, 0x25, 0x73, 0x00, 0x44, 0x55, 0x50,
        0x4d, 0x53, 0x47, 0x4e, 0x4d, 0x00, 0x44, 0x75,
        0x70, 0x6c, 0x69, 0x63, 0x61, 0x74, 0x65, 0x20,
        0x6d, 0x65, 0x73, 0x73, 0x61, 0x67, 0x65, 0x20,
        0x6e, 0x61, 0x6d, 0x65, 0x20, 0x25, 0x73, 0x00,
        0x44, 0x55, 0x50, 0x4d, 0x53, 0x47, 0x49, 0x44,
        0x00, 0x44, 0x75, 0x70, 0x6c, 0x69, 0x63, 0x61,
        0x74, 0x65, 0x20, 0x6d, 0x65, 0x73, 0x73, 0x61,
        0x67, 0x65, 0x20, 0x49, 0x44, 0x73, 0x20, 0x66,
        0x6f, 0x72, 0x20, 0x6e, 0x61, 0x6d, 0x65, 0x73,
        0x20, 0x25, 0x73, 0x20, 0x61, 0x6e, 0x64, 0x20,
        0x25, 0x73, 0x00, 0x46, 0x49, 0x4c, 0x43, 0x4c,
        0x4f, 0x53, 0x45, 0x00, 0x45, 0x72, 0x72, 0x6f,
        0x72, 0x20, 0x63, 0x6c, 0x6f, 0x73, 0x69, 0x6e,
        0x67, 0x20, 0x66, 0x69, 0x6c, 0x65, 0x20, 0x25,
        0x73, 0x00, 0x46, 0x49, 0x4c, 0x57, 0x52, 0x49,
        0x54, 0x45, 0x00, 0x45, 0x72, 0x72, 0x6f, 0x72,
        0x20, 0x77, 0x72, 0x69, 0x74, 0x69, 0x6e, 0x67,
        0x20, 0x66, 0x69, 0x6c, 0x65, 0x20, 0x25, 0x73,
        0x00, 0x42, 0x41, 0x44, 0x54, 0x4f, 0x4b, 0x45,
        0x4e, 0x00, 0x43, 0x61, 0x6e, 0x6e, 0x6f, 0x74,
        0x20, 0x70, 0x61, 0x72, 0x73, 0x65, 0x20, 0x74,
```

```
        0x6f, 0x6b, 0x65, 0x6e, 0x20, 0x5c, 0x22, 0x25,
        0x73, 0x5c, 0x22, 0x00, 0x42, 0x41, 0x44, 0x4e,
        0x50, 0x41, 0x52, 0x4d, 0x00, 0x54, 0x6f, 0x6f,
        0x20, 0x25, 0x73, 0x20, 0x70, 0x61, 0x72, 0x61,
        0x6d, 0x65, 0x74, 0x65, 0x72, 0x73, 0x20, 0x74,
        0x6f, 0x20, 0x25, 0x73, 0x20, 0x63, 0x6f, 0x6d,
        0x6d, 0x61, 0x6e, 0x64, 0x00, 0x42, 0x41, 0x44,
        0x46, 0x4c, 0x4c, 0x45, 0x4e, 0x00, 0x25, 0x73,
        0x20, 0x74, 0x6f, 0x6f, 0x20, 0x73, 0x68, 0x6f,
        0x72, 0x74, 0x20, 0x74, 0x6f, 0x20, 0x62, 0x65,
        0x20, 0x74, 0x68, 0x65, 0x20, 0x6e, 0x61, 0x6d,
        0x65, 0x20, 0x6f, 0x66, 0x20, 0x61, 0x20, 0x2e,
        0x6d, 0x73, 0x67, 0x20, 0x66, 0x69, 0x6c, 0x65,
        0x00, 0x42, 0x41, 0x44, 0x46, 0x4c, 0x4e, 0x41,
        0x4d, 0x00, 0x25, 0x73, 0x20, 0x64, 0x6f, 0x65,
        0x73, 0x20, 0x6e, 0x6f, 0x74, 0x20, 0x65, 0x6e,
        0x64, 0x20, 0x77, 0x69, 0x74, 0x68, 0x20, 0x66,
        0x69, 0x6c, 0x65, 0x20, 0x65, 0x78, 0x74, 0x65,
        0x6e, 0x73, 0x69, 0x6f, 0x6e, 0x20, 0x2e, 0x6d,
        0x73, 0x67, 0x00, 0x42, 0x41, 0x44, 0x46, 0x4c,
        0x46, 0x4d, 0x54, 0x00, 0x25, 0x73, 0x20, 0x69,
        0x73, 0x20, 0x6e, 0x6f, 0x74, 0x20, 0x61, 0x20,
        0x2e, 0x6d, 0x73, 0x67, 0x20, 0x66, 0x69, 0x6c,
        0x65, 0x00, 0x51, 0x55, 0x4c, 0x4e, 0x45, 0x47,
        0x41, 0x54, 0x00, 0x51, 0x75, 0x61, 0x6c, 0x69,
        0x66, 0x69, 0x65, 0x72, 0x20, 0x2d, 0x2d, 0x25,
        0x73, 0x20, 0x69, 0x73, 0x20, 0x6e, 0x6f, 0x74,
        0x20, 0x6e, 0x65, 0x67, 0x61, 0x74, 0x61, 0x62,
        0x6c, 0x65, 0x00, 0x51, 0x55, 0x4c, 0x4e, 0x4f,
        0x56, 0x41, 0x4c, 0x00, 0x51, 0x75, 0x61, 0x6c,
        0x69, 0x66, 0x69, 0x65, 0x72, 0x20, 0x2d, 0x2d,
        0x25, 0x73, 0x20, 0x61, 0x6c, 0x77, 0x61, 0x79,
        0x73, 0x20, 0x74, 0x61, 0x6b, 0x65, 0x73, 0x20,
        0x61, 0x20, 0x76, 0x61, 0x6c, 0x75, 0x65, 0x00,
        0x51, 0x55, 0x4c, 0x55, 0x4e, 0x44, 0x45, 0x46,
        0x00, 0x51, 0x75, 0x61, 0x6c, 0x69, 0x66, 0x69,
        0x65, 0x72, 0x20, 0x25, 0x73, 0x20, 0x6e, 0x6f,
        0x74, 0x20, 0x72, 0x65, 0x63, 0x6f, 0x67, 0x6e,
        0x69, 0x7a, 0x65, 0x64, 0x00, 0x51, 0x55, 0x4c,
        0x56, 0x41, 0x4c, 0x55, 0x44, 0x00, 0x51, 0x75,
        0x61, 0x6c, 0x69, 0x66, 0x69, 0x65, 0x72, 0x20,
        0x2d, 0x2d, 0x25, 0x73, 0x20, 0x6e, 0x65, 0x76,
        0x65, 0x72, 0x20, 0x74, 0x61, 0x6b, 0x65, 0x73,
        0x20, 0x61, 0x20, 0x76, 0x61, 0x6c, 0x75, 0x65,
        0x00
    };

    /*------------------------------------------------------------------------*//*!
    \var   static const MSG_MsgList MSG_MsgList_M2S
    \brief Formatting strings for message facility M2S
    *//*------------------------------------------------------------------------*/
    static const MSG_MsgList MSG_MsgList_M2S[17] =
    {
        { 0x0000ccea, 0, 7,  7, 0, 0x00000000, 0x00000000, 0x00000008 },
        { 0x0002a505, 3, 7, 21, 1, 0x00000000, 0x00000010, 0x00000018 },
        { 0x00033498, 3, 7, 29, 1, 0x00000000, 0x0000002e, 0x00000036 },
```

```
        { 0x00050e1c, 3, 8, 31, 2, 0x00000001, 0x00000054, 0x0000005d },
        { 0x00095576, 3, 8, 25, 1, 0x00000000, 0x0000007d, 0x00000086 },
        { 0x0009559a, 3, 8, 41, 2, 0x00000000, 0x000000a0, 0x000000a9 },
        { 0x000a94a6, 3, 8, 21, 1, 0x00000000, 0x000000d3, 0x000000dc },
        { 0x000aa3da, 3, 8, 21, 1, 0x00000000, 0x000000f2, 0x000000fb },
        { 0x000ca465, 3, 8, 25, 1, 0x00000000, 0x00000111, 0x0000011a },
        { 0x000cb016, 3, 8, 31, 2, 0x00000000, 0x00000134, 0x0000013d },
        { 0x000cbf05, 3, 8, 42, 1, 0x00000000, 0x0000015d, 0x00000166 },
        { 0x000cbf2e, 3, 8, 40, 1, 0x00000000, 0x00000191, 0x0000019a },
        { 0x000cc0bb, 3, 8, 21, 1, 0x00000000, 0x000001c3, 0x000001cc },
        { 0x000e935f, 3, 8, 31, 1, 0x00000000, 0x000001e2, 0x000001eb },
        { 0x000e9648, 3, 8, 35, 1, 0x00000000, 0x0000020b, 0x00000214 },
        { 0x000ea63b, 3, 8, 27, 1, 0x00000000, 0x00000238, 0x00000241 },
        { 0x000eb11e, 3, 8, 34, 1, 0x00000000, 0x0000025d, 0x00000266 }
};

/*-------------------------------------------------------------------------*//*!
\var   static const MSG_FacList MSG_FacList_M2S
\brief Facility header structure for message facility M2S
*//*-------------------------------------------------------------------------*/
static const MSG_FacList MSG_FacList_M2S =
{
    MSG_StrList_M2S, 253, 3,   17, &MSG_MsgList_M2S[0], "M2S"
};

void _GLOBAL__I_MSG_insertFacility_M2S() __attribute__ ((constructor));
void _GLOBAL__D_MSG_removeFacility_M2S() __attribute__ ((destructor));

/*-------------------------------------------------------------------------*//*!
\fn    void _GLOBAL__I_MSG_insertFacility_M2S
\brief Static constructor to insert message facility M2S

Static constructor to insert facility M2S into the message facility list
*//*-------------------------------------------------------------------------*/
void _GLOBAL__I_MSG_insertFacility_M2S()
{
    MSG_insertFacility( &MSG_FacList_M2S );
    return;
}

/*-------------------------------------------------------------------------*//*!
\fn    void _GLOBAL__D_MSG_removeFacility_M2S
\brief Static destructor to remove message facility M2S

Static destructor to remove facility M2S from the message facility list
*//*-------------------------------------------------------------------------*/
void _GLOBAL__D_MSG_removeFacility_M2S()
{
    MSG_removeFacility( &MSG_FacList_M2S );
    return;
}

#ifdef __cplusplus
}
#endif
```

Figure 2 The file `MSG_msgs.c`

```
    /*------------------------------------------------------------------------*//*!
    \file  M2S_msgs.h
    \brief Message definitions for message facility \b M2S (ID: \c 253, \c 0xfd)
    \warning Machine generated code - NEVER edit by hand

    M2S_msgs.h provides all the \c #define statements for the codes in message
    facility \b M2S.  The following table provides the equivalences between
    the \c #define name, the \c #define value and the associated formatting
    string.

    <table>
    <tr>
      <td>Name</td>
      <td>Value</td>
      <td>Associated formatting string</td>
    </tr>
    <tr>
      <td><tt>M2S_SUCCESS</tt></td>
      <td><tt>0x7e8333a8</tt></td>
      <td><tt>"Success"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_FILOPEN</tt></td>
      <td><tt>0x7e8a9417</tt></td>
      <td><tt>"Error opening file %s"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_BADQUAL</tt></td>
      <td><tt>0x7e8cd263</tt></td>
      <td><tt>"Cannot parse qualifier \"%s\""</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_ALOCFAIL</tt></td>
      <td><tt>0x7e943873</tt></td>
      <td><tt>"Cannot allocate %d bytes for %s"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_DUPMSGNM</tt></td>
      <td><tt>0x7ea555db</tt></td>
      <td><tt>"Duplicate message name %s"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_DUPMSGID</tt></td>
      <td><tt>0x7ea5566b</tt></td>
      <td><tt>"Duplicate message IDs for names %s and %s"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_FILCLOSE</tt></td>
      <td><tt>0x7eaa529b</tt></td>
      <td><tt>"Error closing file %s"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_FILWRITE</tt></td>
      <td><tt>0x7eaa8f6b</tt></td>
      <td><tt>"Error writing file %s"</tt></td>
    </tr>
```

```
    <tr>
      <td><tt>M2S_BADTOKEN</tt></td>
      <td><tt>0x7eb29197</tt></td>
      <td><tt>"Cannot parse token \"%s\""</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_BADNPARM</tt></td>
      <td><tt>0x7eb2c05b</tt></td>
      <td><tt>"Too %s parameters to %s command"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_BADFLLEN</tt></td>
      <td><tt>0x7eb2fc17</tt></td>
      <td><tt>"%s too short to be the name of a .msg file"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_BADFLNAM</tt></td>
      <td><tt>0x7eb2fcbb</tt></td>
      <td><tt>"%s does not end with file extension .msg"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_BADFLFMT</tt></td>
      <td><tt>0x7eb302ef</tt></td>
      <td><tt>"%s is not a .msg file"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_QULNEGAT</tt></td>
      <td><tt>0x7eba4d7f</tt></td>
      <td><tt>"Qualifier --%s is not negatable"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_QULNOVAL</tt></td>
      <td><tt>0x7eba5923</tt></td>
      <td><tt>"Qualifier --%s always takes a value"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_QULUNDEF</tt></td>
      <td><tt>0x7eba98ef</tt></td>
      <td><tt>"Qualifier %s not recognized"</tt></td>
    </tr>
    <tr>
      <td><tt>M2S_QULVALUD</tt></td>
      <td><tt>0x7ebac47b</tt></td>
      <td><tt>"Qualifier --%s never takes a value"</tt></td>
    </tr>
    </table>
    *//*------------------------------------------------------------------------*/

    #ifndef M2S_MSGS_H
    #define M2S_MSGS_H

    #ifndef CMX_DOXYGEN

    #define  M2S_SUCCESS          0x7e8333a8
    #define  M2S_FILOPEN          0x7e8a9417
    #define  M2S_BADQUAL          0x7e8cd263
```

```
#define  M2S_ALOCFAIL         0x7e943873
#define  M2S_DUPMSGNM         0x7ea555db
#define  M2S_DUPMSGID         0x7ea5566b
#define  M2S_FILCLOSE         0x7eaa529b
#define  M2S_FILWRITE         0x7eaa8f6b
#define  M2S_BADTOKEN         0x7eb29197
#define  M2S_BADNPARM         0x7eb2c05b
#define  M2S_BADFLLEN         0x7eb2fc17
#define  M2S_BADFLNAM         0x7eb2fcbb
#define  M2S_BADFLFMT         0x7eb302ef
#define  M2S_QULNEGAT         0x7eba4d7f
#define  M2S_QULNOVAL         0x7eba5923
#define  M2S_QULUNDEF         0x7eba98ef
#define  M2S_QULVALUD         0x7ebac47b

#endif    /* CMX_DOXYGEN */

#endif    /* M2S_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	Steve Mazzoni
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	Ed Costello
10	160-175	0xa0-0xaf	Ed Bacho
11	176-191	0xb0-0xbf	Don May
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
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 const char
2          rtn[] = “my_routine”; /* See subsequent examples for a better method */
       .
       .
       .
       if( (*buffer = malloc( size )) == NULL )
       {
3          status = MSG_report( MSG_ALOCFAIL, 0, rtn, 1, size );
4          return( status );
       }
       .
       .
       .
5      return( MSG_SUCCESS );
   }
```

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

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 reported message captures the routine name as part of the message, so please provide (an accurate) one. There are simple ways to do this. Please see section 2.2 where this question is dealt with in more detail.
3. The heart of the system. The first four arguments are mandatory:
   - A message code.
   - A time-stamp. This variable is of type `WCT_time` (as defined by the PBS package). This can be used to associate a specific time with the message. If coded as zero (as was done here), `MSG_report()` will attach its own concept of the "current time".
   - The routine name.
   - The number of parameters that follow. This will be dealt with in more detail in section 2.6.
   - Subsequent arguments are variadic in the same way that arguments to `printf` are variadic. If you look back at the definition of the code `MSG_ALOCFAIL`, you will see that it's expecting a single integer argument.

   The return code reflects the message code passed in as the first argument *with the most significant bit set if the message was successfully reported*. Thus in this example, it is extremely unlikely that `(status == MSG_ALOCFAIL)`. The value of this "reported" bit is that, as a message code is returned through the call stack, subsequent efforts to `MSG_report()` the value will not generate another message.

Warning: the fact that `MSG_report()` can (and usually does) return a status value not equal to the message code can make testing the return value hazardous. Please pay special attention in the next section on how to receive and deal with message codes.

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 "reported" flag set.
5. `MSG_SUCCESS` is not reported, 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;

    .
    .
    .
    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 most significant bit set to indicate that it’s already been reported. The value will *not* be equal to `MSG_ALOCFAIL`.

I realize that I’m harping on about this “reported” 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 reported” bit).

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;

    .
    .
    .
    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 reports 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 report this too:

```
#include “MSG/MSG_msgs.h”

unsigned int my_caller()
{
    unsigned int;
        status;

    char
       *buffer;

    .
    .
    .
    status = my_routine( 1000, &buffer );
    if( _msg_failure( status ) )
    {
        status = _msg_report( MSG_MOREFAIL, 0, 1, <more arguments>  );
        return( status );
    }
    .
    .
    .
    return( MSG_SUCCESS );
}
```

Figure 7 Using `MSG_report()` 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.

This example also introduces the recommended method for passing the routine name. The macro `_msg_report` will thunk its arguments into a call to `MSG_report()`, inserting the routine name via the compiler provided `__func__` macro. For more details, see section 2.2.

2.2 **`MSG_report()`** and **`_msg_report`**

The true entry point provided by MSG for reporting is `MSG_report()`. The standard third call parameter to `MSG_report()` is the name of the current routine. This can be a laborious parameter to maintain “longhand”. The compiler (more accurately the preprocessor) can help out with the maintenance of this parameter via its `__func__` macro. Wherever this macro appears, the preprocessor will substitute a reference to the name of the current routine. It’s as if the compiler inserted the following code at the top of every routine where the `__func__` macro appears:

```
unsigned int <my_routine>()
{
    static const char __func__[] = “<my_routine>”;
        status;
    .
    .
    .
}
```

Figure 8 Pseudo-code representing the action of the `__func__` macro

Life is immediately much easier. Upgrading a shard of code from a previous example:

```
#include “MSG/MSG_msgs.h”

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

    unsigned int
        status;
    .
    .
    .
    if( (*buffer = malloc( size )) == NULL )
    {
        status = MSG_report( MSG_ALOCFAIL, 0, __func__, 1, size );
        return( status );
    }
    .
    .
    .
    return( MSG_SUCCESS );
}
```

Figure 9 Using the __func__ macro in a MSG_report() call

The macro _msg_report just takes this mechanism to its logical conclusion. It will auto-supply the __func__ macro into a simpler call:

```
#include “MSG/MSG_msgs.h”

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

    unsigned int
        status;
    .
    .
    .
    if( (*buffer = malloc( size )) == NULL )
    {
        status = _msg_report( MSG_ALOCFAIL, 0, 1, size );
        return( status );
    }
    .
    .
    .
    return( MSG_SUCCESS );
}
```

Figure 10 Using the _msg_report() macro to automate the inclusion of the current routine name

In this version, the caller no longer has to deal with the routine name at all. Note however that because `__func__` is a mechanism belonging to the preprocessor, this trick is only possible with another preprocessor macro. For the curious, the actual macro looks like:

```
#define  _msg_report( code, time,           nprm,     args... ) \
          MSG_report( code, time, __func__, nprm , ## args    )
```

Figure 11 Definition of the `_msg_report` macro

Historically, the compiler has also provided the `__FUNCTION__` macro with similar, but not quite identical characteristics to the `__func__` macro (the technical difference is that `__FUNCTION__` behaved like a string literal where `__func__` behaves like a constant string variable). Please don't use the `__FUNCTION__` macro. Its use has been deprecated in the `gcc` (and other) compiler suite(s). Besides, why would you *not* want to use the `_msg_report()` macro?

2.3 Reporting Levels

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

Without intervention, the `MSG_report()` 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;

    .
    .
    /* Lots of very complicated initialization                    */
    .
    .
    /* Message code MSG_INITGOOD has severity level S (success)   */
    status = _msg_report( MSG_INITGOOD, 0, 0 );
    return( status );
}
```

Figure 12 Unsuccessful attempt to report a success message

A temporary redefinition of the MSG default reporting 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;

    .
    .
    /* Lots of very complicated initialization                    */
    .
    .
    /* Message code MSG_INITGOOD has severity level S (success)   */
    level  =  MSG_setLevel( MSG_LVL_SUCCESS );
    status = _msg_report( MSG_INITGOOD, 0, 0 );
    level  =  MSG_setLevel( level           );
    return( status );
}
```

Figure 13 Successful attempt to report a success message

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

2.4 Trace Buffers

Analogous to the level setting system is the trace buffer system. Every message generated by `MSG_report()` 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;


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

Figure 14 Attaching a trace buffer number to a message

2.5 Reporting 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 reporting levels and trace buffers numbers.

Reporting levels and trace buffer numbers are maintained *on a per thread/task basis*. That means that if task A sets the reporting level down to information, but gets interrupted by task B before it can report, task B does *not* inherit the reporting level from task A. Similarly for trace buffer numbers.

2.6 Reporting With Partial Formatting

The fourth argument to `MSG_report()` 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 reporting level, even if the called routine in turn calls `MSG_report()`, 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 report it. The first instinct is to use the `MSG_setLevel()` routine to drop the reporting level to warning and then report 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 reported with *partial* formatting. That is the real purpose of the fourth argument to `MSG_report()` (third argument to `_msg_report()`). The fourth 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 fourth argument to `MSG_report()` 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`.

2.7 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_report()`

And the number of macros is even smaller:

- `_msg_success`
- `_msg_failure`
- `_msg_match`
- `_msg_report()`

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 15 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(s) are 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(s) are 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(s) from the "message compilation" step are used in a regular constituent. The only trick here is that the constituent that produces the `.c` file(s) (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.

CMX has recently added some functionality to help out in this situation (CMX `V2-0-9` to be precise). Whenever the `msg2src` step produces a derived file, CMX will create/edit a `.cvsignore` file in the appropriate directory and list the derived file there. CMX will also *remove* the relevant `.cvsignore` file entries when a "`document msg`" constituent is cleaned.

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!)

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_report()`. 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_report()`. 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 section 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:

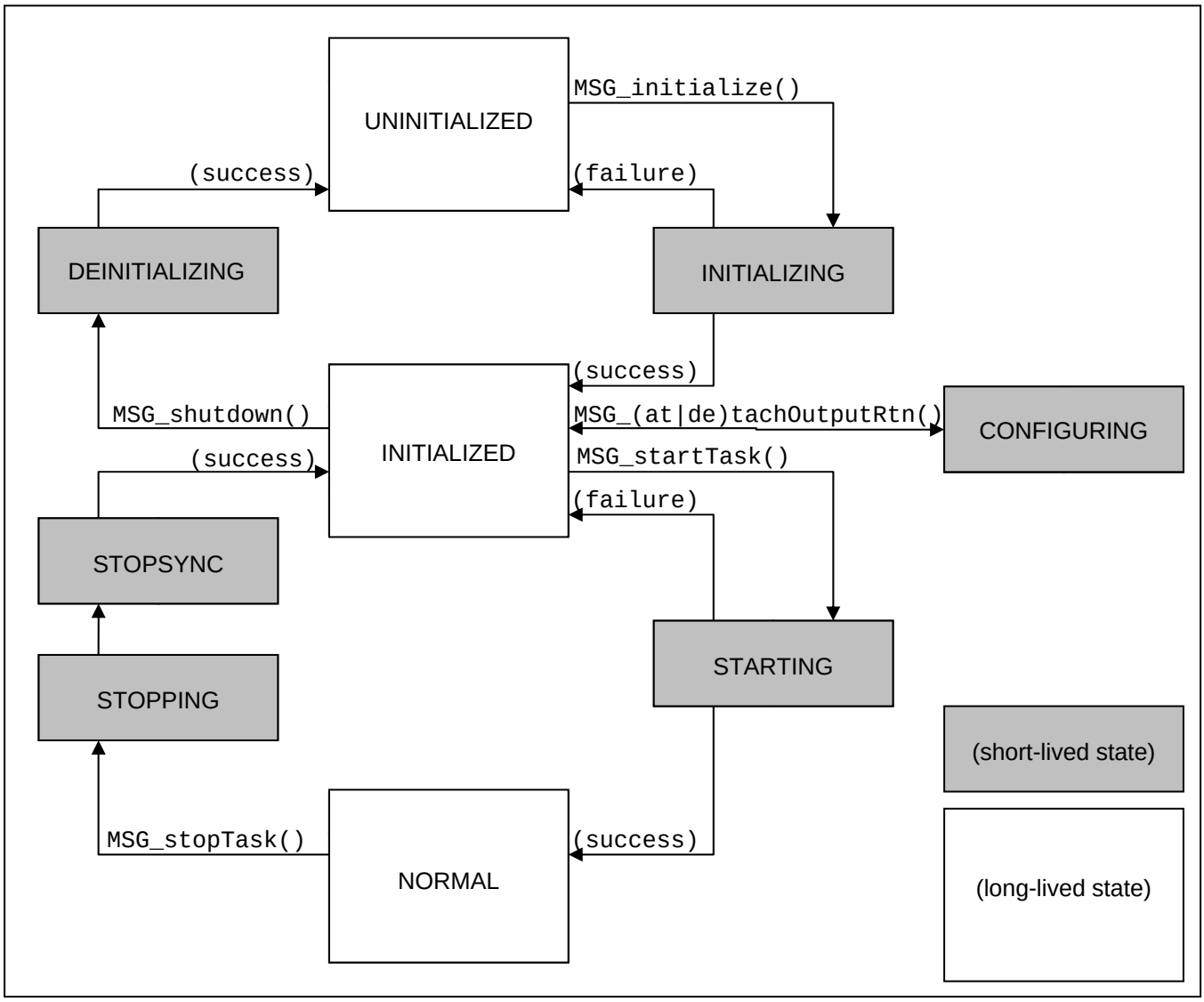

Figure 16 Starting and stopping the message system.

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.

If a `NULL` pointer is passed to `MSG_initialize()`, it will make default allocations (currently 32 packets, each 256 bytes long). With this allocation, 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 cans through all attached output processors making an initialization call to each of them.
- It spawns the message task (using the PBS facility FORK).

Like `MSG_initialize()`, `MSG_startTask()` takes a parameter block which it uses to set up the task. If a `NULL` pointer is passed to `MSG_startTask()`, it will construct a default parameter block. If a genuine parameter block is passed to MSG_startTask(), 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 17 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 18 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 19 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 20 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_report()` 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_report()` 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 report messages. The life cycle of a message packet is depicted in the following figure:

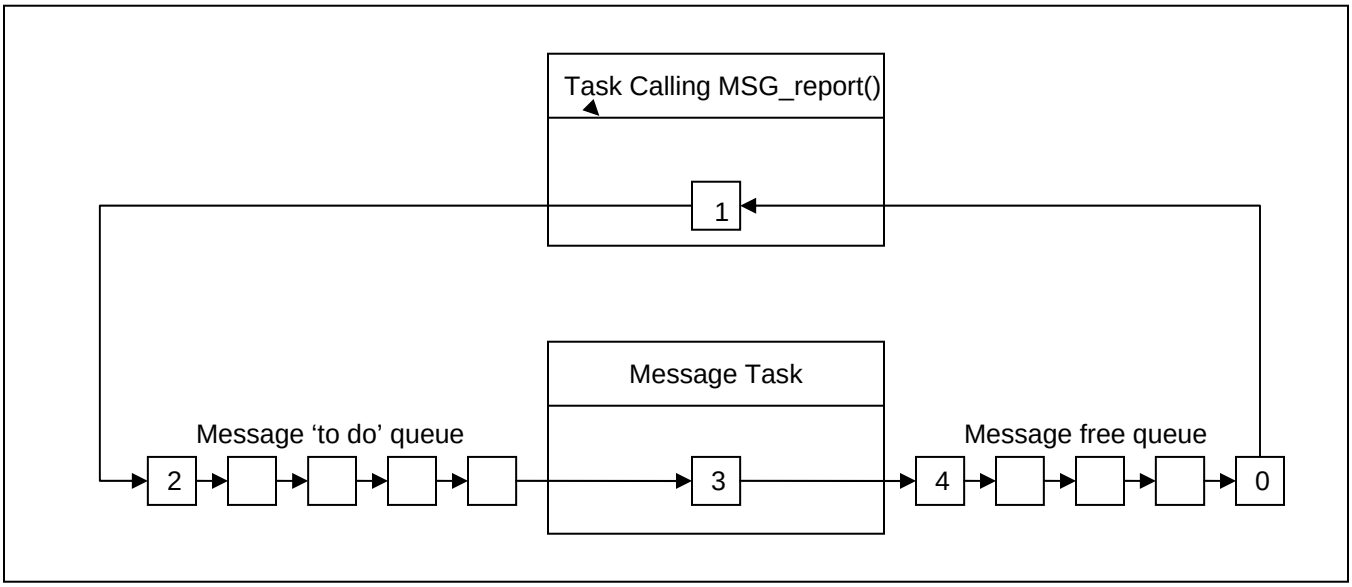

Figure 21 Life cycle of a message packet

0. When a task calls `MSG_report()`, `MSG_report()` allocates a message packet from the message free queue.
1. `MSG_report()` captures information, including the variadic parameter list, into the packet.
2. `MSG_report()` 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 report 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 report can't allocate a packet from the free list, it will instead use the first of the lookaside packets to report a "blackout begin" message. All messages reported 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 report a "blackout end" message (this message includes a count of dropped messages) and re-enables reporting.

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 reporting 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_report()` from interrupt level will not cause the system to die.

It's another question whether it is wise to call `MSG_report()` from interrupt level. The cost is increased interrupt latency if `MSG_report()` 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_report()` 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_report()` 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 22 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 23 Stopping/shutting down the message system.

5 Tips, Tricks and Restrictions

5.0 What Severity Code and When To Report

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

- Report *all* messages of any severity.
- Reduce the reporting level to `MSG_LVL_INFORMATION` globally.

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

- Only report error severity messages.
- Keep the reporting 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 reporting 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).
- Report any `ERROR` or `WARNING` severity messages immediately upon detection. Reporting `SUCCESS` or `INFORMATION` severity messages is certainly allowed but not common.
- For special `SUCCESS` or `INFORMATION` severity messages that should be reported, drop the reporting 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 24 The minimalist `.msg` file

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

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.

5.2 Restrictions

5.2.0 Thread Safety

Considerable effort has gone into this version of MSG to ensure thread safety for all the control structures and lists. There is however one exception:

The entry point `MSG_initialize()` is not thread safe.

The reason `MSG_initialize()` remains non-thread safe is the classic bootstrap problem of how to lock around the allocation of the locking mechanism. There are ways to do this, but the cost is quite high, and the administrative fiat that calls to `MSG_initialize()` be organized to be thread safe should be sufficient.

5.2.1 Reference Semantics for Copy Semantics

To improve efficiency, this version of message has adopted reference semantics instead of copy semantics for the capture of the task name and the routine name during a call to `MSG_report()`. This means that the string references must continue to be valid at the time they are dereferenced (in the message task). Note that this only applies to the task and routine strings. Strings provided as parameters to the message formatting continue to be copied.

5.2.1.0 Task Name

The user does not have any direct control over the capture of the task name. The only time when a reference to the task name, instead of a copy of the task name, impinges on the user's

existence is the rare case that the task dies between the moment the message is captured (during `MSG_report()`) and the time its processed (in the message task). In these cases, the task name might be garbage (though the message task will at least ensure that the string has coherence, e.g. it will guarantee that the string is null-terminated after, at most, sixteen characters).

5.2.1.1 Routine Name

The caller of `MSG_report()` *does* have direct control over the routine name, which means that the caller is responsible for ensuring that the routine name is persistent. The recommended method of calling `MSG_report()` using the `_msg_report` macro will guarantee the correct persistence model.

5.2.2 Tasks Not Started With The TASK Routines in Package PBS

The methods used to identify a task's reporting level and trace buffer number are based on the "task block of data" (TBD) facility in package PBS. Tasks not started with the TASK routines in PBS do not have a TBD block. Typical examples of such tasks are:

- The "main" thread of execution in a multi-threaded host application.
- The shell task (or its Tornado surrogate) in VxWorks (embedded) systems.

These tasks do *not* honour requests to change the reporting level or the trace buffer number. If it's an issue, a TBD block can be added to such a task with a call to the `TBD_convert()` function (see package PBS for details).

6 Deprecated

Three MSG entry points are now deprecated. All continue to function in this release, but a release will come where they are no longer supported.

6.0 MSG_signal()

This is the one everyone will want to kill me for!

The shortcomings of this interface came up at a code review. Once pointed out, I kicked myself for not having seen it myself. There was no need to define an in/out argument to this routine and ugly up everyone's code to support it. Having decided to replace the interface, I went on to

- Expose the message time-stamp so that users can define their own if they want to.
- Surround the MSG_report() call with the macro _msg_report() to make life easier

In terms of ugliness, compare the following to code snippets:

```
if( (*buffer = malloc( size )) == NULL )
{
    status = MSG_ALOCFAIL;
    status = MSG_signal( &status, __func__, 1, size );
    return( status );
}
```

Figure 25 Message call using `MSG_signal()`

```
if( (*buffer = malloc( size )) == NULL )
{
    status = _msg_report( MSG_ALOCFAIL, WCT_get(), 1, size );
    return( status );
}
```

Figure 26 Message call using `_msg_report()`

`MSG_signal()` has therefore been replaced with `MSG_report()`.

The backward compatibility support for this is a little byzantine. `MSG_signal()` is now defined as a preprocessor macro that simply thunks its arguments to the `MSG_report()` format. To support existing binaries, an entry point `MSG_signal()` does still exist, but is now inaccessible at compile time. My goal here is to migrate packages to the `MSG_report()` interface with no intervention from developers, though developers are encouraged to retrofit `_msg_report()` calls opportunistically.

6.1 MSG_start()

The parameters defined for this entry point weren't sufficiently rich to support the management of the message task. This entry point has been replaced with `MSG_startTask()`.

6.2 MSG_stop()

This entry point was superceded for reasons of symmetry. It didn't make sense (to me) to have a pair of routines called `MSG_startTask()` and `MSG_stop()`. This entry point has been replaced with `MSG_stopTask()`.

7 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.

7.0 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!)

7.1 Common Run Time Improvements

7.1.0 CPU Identification

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

7.2 Single-Threaded Specific Run Time Improvements

7.3 Multi-Threaded Specific Run Time Improvements

7.4 Ground Based Additions

7.4.0 A Logging Database in ISOC

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>”.

7.4.1 A Message Database in 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.