SUN/104.12

Starlink Project
Starlink User Note 104.12

P C T Rees
A J Chipperfield
T Jenness

23 July 2009

MERS (MSG and ERR) Message and Error Reporting Systems Version 2.1 Programmer's Manual

Abstract

This document describes two C subroutine libraries, MSG and ERR, which can be used to provide informational text to the user from any application program. It also describes the Fortran interface for the two libraries.

The Message Reporting System, MSG, is used for reporting non-error information and the Error Reporting System, ERR, is used specifically for reporting error messages. The conventions for reporting errors from Starlink software are also discussed in detail.

This document is recommended reading for anyone writing applications software for use on Starlink.

Contents

List of Figures

1 Introduction

There is a general need for application programs on Starlink to provide the user with informative textual messages about:

- What they do – for example, during long operations it is helpful if the user is kept informed of what a program is doing.
- What results have been obtained – for example, the notification of the final results from a procedure, or of some intermediate results that would help the user respond to further prompts.
- What errors have occurred – for example, errors which lead to the user being prompted to provide more sensible input to a program, or fatal errors which cause an application to stop.

This document describes two subroutine libraries which can be used to provide informational text to the user from an application program. The two packages are:

MSG Message Reporting System, used for reporting non-error information.

ERR Error Reporting System, used specifically for reporting error messages.

This document is aimed at all programmers who are writing applications software on Starlink, either for use within the ADAM environment (see SG/4) or stand-alone. The major part of this document concerns the use of the stand-alone version of the Message and Error Reporting System subroutines. Its use in ADAM applications is essentially the same but there are some additional features which are described in Appendix E.

2 MSG – Message Reporting System

2.1 Overview

The most obvious way of producing informational messages from within Fortran or C application programs is through formatted `printf`, `WRITE` and `PRINT` statements. However:

- It is generally considered a good idea in a large system to direct all output through a single routine to improve portability and to make re-direction of output easier.
- Some environments, such as ADAM, require output via a 'user-interface' program and not direct to the terminal.
- It is sometimes difficult to format numerical output in its most concise form within textual messages. To do this in-line each time a message is sent to the user would be very inconvenient and justifies the provision of a dedicated set of subroutines.

These considerations have led to the design and implementation of a set of subroutines which form the Message Reporting System. The Message System subroutines have names of the form

```
msgName     [C]
MSG_name    [Fortran]
```

where *name* indicates what the subroutine does and follows the standard Starlink naming convention for C and Fortran. This document will provide examples in both languages, but subroutine names in the text will use the C naming convention.

2.2 Reporting messages

The primary message reporting subroutine is `msgOut`. It has a calling sequence of the form:

```
msgOut( param, text, status );

CALL MSG_OUT( PARAM, TEXT, STATUS )
```

where the argument *param* is a character string giving the name of the message, *text* is a character string giving the message text, and *status* is the integer subroutine status value (a pointer to an int in C). `msgOut` sends the message string, *text*, to the standard output stream which will normally be the user's terminal.

The subroutine `msgOut` uses the Starlink convention of inherited status. This means that calls to `msgOut` will not output the message unless the given value of *status* is equal to `SAI__OK`. If an error is encountered within the subroutine, then *status* is returned set to an error value. (The global constant `SAI__OK` is defined in the include file *sae_par.h* (*SAE_PAR* for Fortran). The use of this global constant and of inherited status are discussed in detail in §3.2.)

The maximum length for an output message is 300 characters. If it exceeds this length, then the message is truncated with an ellipsis, *i.e.* "...", but no error will result.

By default, messages are split so that output lines do not exceed 79 characters – the split is made on word boundaries if possible. The maximum output line size can be altered using tuning parameters (see §4).

It is recommended that, within the application, the message name, *param*, should be a unique identifier for the message string, *text*. However, the message name serves no useful purpose within a stand-alone application and is often given as a blank string. In ADAM applications the message name has a specific purpose which is discussed in detail in Appendix E.2.

Here is an example of using `msgOut`:

```
msgOut( "EXAMPLE_MSGOUT", "An example of msgOut", status );

CALL MSG_OUT( 'EXAMPLE_MSGOUT', 'An example of MSG_OUT.', STATUS )
```

It is sometimes useful to intersperse blank lines amongst lines of textual output for clarity. This could be done using calls to `msgOut`, *e.g.*

```
msgOut( "MSG_BLANK", "", status );

CALL MSG_OUT( 'MSG_BLANK', ' ', STATUS )
```

For convenience, the subroutine `msgBlank` has been provided for this purpose, *e.g.*

```
msgBlank( status );

CALL MSG_BLANK( STATUS )
```

The status argument in `msgBlank` behaves in the same way as the status argument in `msgOut`.

2.3 Conditional message reporting

It is sometimes useful to have varying levels of message output which may be controlled by the user of an application. Instances where this facility might be of use are:

- the ability to switch on informational messages for novice users which can later be switched off as the user becomes familiar with the application;
- the ability to switch off unnecessary informational messages when an application is run in batch mode, or within a command procedure;
- the ability to output detailed information from an application on request, say from within an iterative procedure.

Conditional message output is achieved explicitly in the Message Reporting System using the subroutine `msgOutif` to assign a "priority" to the message, *e.g.*

```
msgOutif( MSG__NORM, "", "A conditional message", status );

CALL MSG_OUTIF( MSG__NORM, ' ', 'A conditional message', STATUS )
```

Here, the first argument is the "priority" associated with the message and can be any one of twenty four levels which are represented by symbolic constants defined in the include file *msg_par.h* (*MSG_PAR*) (see §6):

`MSG__QUIET` – quiet mode, high priority;

`MSG__NORM` – normal mode, normal priority (default);

`MSG__VERB` – verbose mode, low priority.

`MSG__DEBUG` – debug mode, lowest priority

`MSG__DEBUGnn` – multiple debug modes. 1 to 20.

Whether or not the message will be output depends upon the "conditional message output filter" which may be set using the subroutine `msgIfset`. *e.g.*

```
msgIfset( MSG__QUIET, status );

CALL MSG_IFSET( MSG__QUIET, STATUS )
```

The first argument of `msgIfset` is the required conditional output filter level – it may take the same values as the message priority with the addition of `MSG__NONE` and `MSG__ALL`; by default it is set to `MSG__NORM`. The current conditional output filtering level may be inquired using subroutine `msgIflev` or compared against using `msgFlevok`. `MSG__NONE` and `MSG__ALL` allow every message to be hidden or all messages to be displayed respectively. These levels can not be used with `msgOutif`.

See also `msgTune` and `msgIfgetenv` that allows the filter level to be set by an environment variable, and `msgIfget` which allows ADAM programs to obtain the filter level from an ADAM parameter.

The action of `msgOutif` resulting from each of the defined priority values is as follows:

`MSG__QUIET` – output the given messsage unless the current output filter is set to `MSG__NONE`;

`MSG__NORM` – output the given message if the current output filter is set to either `MSG__NORM`, `MSG__VERB`, `MSG__DEBUG` (and related constants) or `MSG__ALL`;

`MSG__VERB` – output the given message only if the current output filter is set to `MSG__VERB`, `MSG__DEBUG` (and related constants) or `MSG__ALL`.

`MSG__DEBUG` – output the given message only if the current output filter is set to `MSG__DEBUG` (and related constants) or `MSG__ALL`.

In this scheme, messages given the priority `MSG__QUIET` the most important messages being output by an application and can only be turned off if the user is forcing all output to be disabled.

Here is an example of how conditional message output might be used in an application using interactive graphics with differing levels of informational messages to match how familiar the user is with the application:

```
*  Use the cursor to enter the approximate positions of stars on the
*  displayed image to be fitted.
      CALL MSG_OUT( ' ', 'Use the cursor to enter star positions',
     :             STATUS )

*  Explain the positioning of the cursor.
      CALL MSG_OUTIF( MSG__NORM, ' ',
     :               'The graphics cursor should be positioned ' //
     :               'close to the centre of each star image', STATUS )

*  Explain the cursor keys to new user.
      CALL MSG_OUTIF( MSG__VERB, ' ', 'Cursor keys: 1 add entry', STATUS )
      CALL MSG_OUTIF( MSG__VERB, ' ', '             2 reject last entry',
     :               STATUS )
      CALL MSG_OUTIF( MSG__VERB, ' ', '             3 entry complete',
     :               STATUS )
```

```
/*  Use the cursor to enter the approximate positions of stars on the
 *  displayed image to be fitted. */
      msgOut( "", "Use the cursor to enter star positions", status );

/*  Explain the positioning of the cursor. */
      msgOutif( MSG__NORM, "",
                "The graphics cursor should be positioned "
                "close to the centre of each star image", status );

/*  Explain the cursor keys to new user. */
```

```
msgOutif( MSG__VERB, "", "Cursor keys: 1 add entry", status );
msgOutif( MSG__VERB, "", "             2 reject last entry", status );
msgOutif( MSG__VERB, "", "             3 entry complete", status );
```

The `msgBlankif` function works in much the same way as `msgOutif` to allow a blank line to be output conditionally.

```
msgBlankif( MSG__VERB, status );

CALL MSG_BLANKIF( MSG__VERB, STATUS )
```

The `MSG__NONE` and `MSG__ALL` levels cannot be used with `msgBlankif`.

2.4 Conditional Message Reporting and `msgOut`

Although conditional message reporting may be handled explicitly by calling `msgOutif`, calls to `msgOut` and `msgBlank` also have conditional message reporting built in. Their output has priority `MSG__NORM` associated with it and will therefore only be output when the conditional output filter is set to `MSG__NORM` (the default), `MSG__VERB` or greater.

2.5 Message tokens

In the previous examples of `msgOut` and `msgOutif`, the message text is "constant" in the sense that it does not refer to any variable items, *e.g.* file names or numeric values. However, very often, applications need to include the values of variables within output messages. This is done in the Message System using tokens embedded within the message text. For example, a program which measures the intensity of an emission line in a spectrum can output its result by:

```
 msgSetr( "FLUX", flux );
 msgOut( "EXAMPLE_RESULT",
         "Emission flux is ^FLUX (erg/cm2/A/s).", status);

 CALL MSG_SETR( 'FLUX', FLUX )
 CALL MSG_OUT( 'EXAMPLE_RESULT',
:             'Emission flux is ^FLUX (erg/cm2/A/s).',
:             STATUS )
```

Here, the subroutine `msgSetr` is called to define a token named "FLUX" and assign to it the value of the `REAL` or `float` variable FLUX encoded as a character string. The token name, immediately preceded by the up-arrow, "^", escape character is then included in the given message text. As the given text is processed, the token is expanded to the string assigned to the token.

For example, If the variable FLUX in this example has the value 2.4, then the message output to the terminal would be:

```
Emission flux is 2.4 (erg/cm2/A/s).
```

There is a set of `msgSet`*x* subroutines, one subroutine for each of five standard Fortran 77 data types (the Fortran type COMPLEX has not been provided for). Here, *x* corresponds to the Fortran data type of the value to be assigned to the named message token:

x	*Fortran Type*
d	DOUBLE PRECISION
r	REAL
i	INTEGER
l	LOGICAL
c	CHARACTER

In each case, the calling sequence is of the form:

```
msgSetx( token, value );

CALL MSG_SETx( TOKEN, VALUE )
```

where *token* is a character string giving the name chosen by the user and *value* is a variable or constant of the appropriate type. The numeric subroutines, `msgSetd`, `msgSetr`, and `msgSeti`, adopt the most concise format that will represent the value by removing trailing zeros, leading and trailing blanks, and by avoiding the use of exponential notation unless it is necessary. `msgSetl` uses `TRUE` or `FALSE` according to the value it is given. `msgSetc` removes trailing blanks from the character string; leading blanks are not removed.

An additional feature of the `msgSet`*x* routines is that calls to these routines using an existing token will result in the value being appended to the previously assigned token string. Here is the previous example written to exploit this feature of the `msgSet`*x* routines:

```
/* Local Constants: */
      const char *funits = " erg/cm2/A/s";
      ...

      msgSetr( "FLUX", flux );
      msgSetc( "FLUX", funits );
      msgOut( "EXAMPLE_RESULT",
              "Emission flux is ^FLUX", status );

*  Local Constants:
      CHARACTER FUNITS * 12
      PARAMETER( FUNITS = ' erg/cm2/A/s' )

      ...

      CALL MSG_SETR( 'FLUX', FLUX )
      CALL MSG_SETC( 'FLUX', FUNITS )
      CALL MSG_OUT( 'EXAMPLE_RESULT',
     :             'Emission flux is ^FLUX.', STATUS )
```

where the CHARACTER variable *funits* has been assigned the value of an appropriate unit of flux (*e.g.* erg/cm2/A/s) earlier in the program. Note that repeated calls to the `msgSet`*x* routines will append values to the token string with no separator, hence a leading space in the *funits* string is needed to separate the flux value and its units in the expanded message.

The text string associated with a message token (*i.e.* the token value) may be up to 200 characters long. Token names may be up to 15 characters long and should be valid names: *i.e.* they should begin with an alphabetic character and continue with alphanumeric or underscore characters. A maximum of 64 uniquely named message tokens may be included in any output message.

No message tokens are defined initially and after each call to `msgOut`, `msgOutif` or `msgLoad` (or a corresponding ERR routine, see §3.4) all existing tokens are left undefined.

2.6 Formatted tokens from C

The `msgSet`*x* functions encode numeric values in the most concise format that will describe the supplied value. Sometimes, however, a specific format is required; for example, the message may form part of a table. More precise control can be obtained by using the `msgFmt` function to provide all the standard sprintf() formats available from the standard C library.

```
msgFmt( token, format, ... );
```

and the previous example could be written as:

```
msgFmt( "FLUX", "%5.2f", flux );
msgSetc( "FLUX", funits );
msgOut( "EXAMPLE_RESULT",
       "Emission flux is ^FLUX", status );
```

There is no restriction on the number of formats that can be included in a single call so the above example can be simplified further:

```
msgFmt( "FLUX", "%5.2f %s", flux, funits );
msgOut( "EXAMPLE_RESULT",
       "Emission flux is ^FLUX", status );
```

2.7 Formatted tokens from Fortran

The Fortran `MSG_SET`*x* subroutines encode numeric values in the most concise format that will describe the supplied value. This is normally what is wanted for a simple message. The Fortran interface provides an API similar to that for the C library but using standard Fortran format conventions.

More precise numeric formats could be achieved using the Fortran `WRITE` statement with the Message System, *e.g.*

```
*  Local Constants:
      CHARACTER * 12 FUNITS
      PARAMETER( FUNITS = ' erg/cm2/A/s' )

*  Local Variables:
      CHARACTER * 12 VALUE

      ...
```

```
*  Output the flux value.
      WRITE( VALUE, '( 1E12.5 )' ) FLUX
      CALL MSG_SETC( 'FLUX', VALUE )
      CALL MSG_SETC( 'FLUX', FUNITS )
      CALL MSG_OUT( 'EXAMPLE_RESULT',
     :             'Emission flux is ^FLUX.', STATUS )
```

which would produce the message:

```
Emission flux is  2.40000E+00 (erg/cm2/A/s).
```

In this case, the first call to `MSG_SETC` assigns the supplied character string *VALUE* to the named token "FLUX".

Since this sequence of a formatted internal `WRITE` followed by a call to `MSG_SETC` is of general use, it is provided in a set of subroutines of the form:

```
CALL MSG_FMTx( TOKEN, FORMAT, VALUE )
```

where *FORMAT* is a valid Fortran 77 format string which can be used to encode the supplied value, *VALUE*. As for `MSG_SET`*x*, *x* corresponds to one of the five standard Fortran data types – *D*, *R*, *I*, *L* and *C*.

Using `MSG_FMT`*x*, the example given above can be performed by:

```
*  Output the flux value.
      CALL MSG_FMTR( 'FLUX', '1E12.5', FLUX )
      CALL MSG_SETC( 'FLUX', FUNITS )
      CALL MSG_OUT( 'EXAMPLE_RESULT',
     :             'Emission flux is ^FLUX.', STATUS )
```

The use of the `MSG_FMT`*x* routines along with their ability to append values of any type to existing tokens is a very powerful tool for constructing tabular output from applications software.

2.8 Direct formatted message output with `msgOut`

The C interface provides variants to the standard `msgOut` and `msgOutif` functions that can handle sprintf() style format strings. This can lead to a significant reduction in the number of lines spent setting up tokens by embedding the formatting directly in the required string. The example from the previous section can simply be written as:

```
msgOutf( "EXAMPLE_RESULT",
       "Emission flux is %5.2f %s", status, flux, funits );
```

`msgOutf` and `msgOutiff` act like `printf()` whereas `msgOutifv` is a variant of `msgOutif` that takes a *va_list* argument rather than variadic "..." arguments. The "%" character is special in this context and the ADAM definition (See Appendix E.2.2) is not available. A literal "%" is obtained by doubling up the symbol ("%%") as would be used in `printf()` and described in the next section.

2.9 Including escape characters in messages

Sometimes it is necessary to include the message token escape character, "^", literally in a message. When the message token escape character is immediately followed by a blank space, or is at the end of the `msgOut` text, it is included literally. If this is not the case, then it can be included literally by duplicating it. So, for example:

```
msgSetc( "TOKEN", "message token" );
msgOut( "EXAM_UPARROW",
        "Up-arrow, ^^, is the ^TOKEN escape character.", status);

 CALL MSG_SETC( 'TOKEN', 'message token' )
 CALL MSG_OUT( 'EXAM_UPARROW',
:             'Up-arrow, ^^, is the ^TOKEN escape character.',
:             STATUS )
```

would produce the message:

```
Up-arrow, ^, is the message token escape character.
```

Escape characters and token names will also be output literally if they appear within the value assigned to a message token; *i.e.* message token substitution is not recursive. This means that if the message system is to be used to output the value of a character variable, the contents of which are unknown and may therefore include escape characters, the value should first be assigned to a message token. Thus,

```
msgSetc( "TEXT", value );
msgOut( "EXAMPLE_OK", "^TEXT", status );

CALL MSG_SETC( 'TEXT', VALUE )
CALL MSG_OUT( 'EXAMPLE_OK', '^TEXT', STATUS )
```

will output the contents of VALUE literally, whereas

```
msgOut( "EXAMPLE_BAD", value, status );

CALL MSG_OUT( 'EXAMPLE_BAD', VALUE, STATUS )
```

might not produce the desired result. This consideration is particularly important when outputting text values such as file names within the ADAM environment, where a number of additional escape characters are defined (see Appendix E.2).

2.10 Intercepting messages

It may sometimes be convenient within an application to write the text of a message, complete with decoded message tokens, to a character variable instead of the standard output stream. The Message System provides subroutine `msgLoad` to do this. `msgLoad` has the calling sequence:

```
msgLoad( param, text, opstr, opstr_len, oplen, status );

CALL MSG_LOAD( PARAM, TEXT, OPSTR, OPLEN, STATUS )
```

Here, the arguments *param*, *text* and *status* are identical to those for `msgOut`. The behaviour of `msgLoad` is also the same as `msgOut` except that, instead of sending the expanded message text to the standard output stream, `msgLoad` returns it in the character variable *optstr* (regardless the output filtering level). *oplen* returns the length of the message in *opstr*. If the message text is longer than the declared length of *opstr* (as specified in *opstr_len* in the C interface), then the message is truncated with an ellipsis, *i.e.* "…", but no error results.

The symbolic constant `MSG__SZMSG` is provided for defining the length of character variables which are to hold such messages. This constant is defined in the include files *msg_par.h* and *MSG_PAR* (see §6).

2.11 Renewing annulled message tokens

Each call to `msgOut`, `msgOutif` or `msgLoad` will annul any defined message tokens, regardless of the success of the call. This feature of the Message Reporting System ensures that message token names can be re-used with safety in a series of calls to, say, `msgOut` (*e.g.* in order to output a table of values line by line). However, under certain circumstances it is useful to be able to restore, or renew, the values of any message tokens set prior to the call to the output routine. This can be done using the subroutine `msgRenew`. In order to be effective, `msgRenew` must be called after the call which annulled the required message tokens and prior to any further message token definitions (*e.g.* using the `msgSet`*x* and `MSG_FMT`*x* routines). If `msgRenew` is called with existing message tokens defined, no action is taken.

Here is a Fortran example of the use of `msgRenew` where a table of values is being output to the user and to a log file:

```
*  Loop to output the table.
      DO 10 I = 1, NROWS

*     Set a token for each column.
         CALL MSG_FMTI( 'COL1', '1X, I10', I )
         CALL MSG_FMTI( 'COL2', 'I7', HDNUMB( I ) )
         CALL MSG_FMTR( 'COL3', 'F10.5', X( I ) )
         CALL MSG_FMTR( 'COL4', 'F10.5', Y( I ) )
         CALL MSG_FMTR( 'COL5', 'F10.5', Z( I ) )

*     Output a row of the table to the user.
         CALL MSG_OUT( ' ', '^COL1 ^COL2 ^COL3 ^COL4 ^COL5', STATUS )

*     Renew the token values.
         CALL MSG_RENEW

*     Build a string with the row of data.
         CALL MSG_LOAD( ' ', '^COL1 ^COL2 ^COL3 ^COL4 ^COL5', OPSTR,
     :                  OPLEN, STATUS )

*     Write the string to a file.
         WRITE( OPFILE, '( 1X, A )', IOSTAT = IOSTAT ) OPSTR( 1 : OPLEN )
 10   CONTINUE
```

2.12 Resilience

Other than `msgOut`, `msgOutif`, `msgBlank`, `msgBlankif` and `msgLoad`, the Message System subroutines do not use a *status* argument. This is because they are intended to be very robust. In order to construct a message for output to the user, they will attempt to recover from any internal failure. The STATUS argument in the "output" routines conforms to the Starlink convention for inherited status (see §3.2). This means that an application can contain sequences of `msgSet`*x*, `MSG_FMT`*x* and `msgOut` calls, and only needs to check the status at the end.

There are two kinds of "failure" that can occur within the Message System:

- Message Construction – Any tokens which cannot be evaluated while constructing the output message are indicated in the message text by using a special syntax. This syntax is illustrated for the text resulting from the call:

  ```
  msgOut( "EXAMPLE_FLUX",  "Emission flux is ^FLUX (erg/cm2/s).",
          status );

  CALL MSG_OUT( 'EXAMPLE_FLUX',
 :             'Emission flux is ^FLUX (erg/cm2/s).', STATUS )
  ```

  If the token "FLUX" is not defined, then this call will produce the text:

  ```
  Emission flux is ^<FLUX> (erg/cm2/s).
  ```

  There are several reasons why a token may be undefined:

  – It has not been defined using `msgSet`*x etc*.
  – It has been annulled by a previous call to `msgOut`, `msgOutif` or `msgLoad` (or a corresponding ERR routine).
  – An attempt has been made to define more than 64 message tokens.
  – An error has been made in a call to one of the `MSG_FMT`*x* subroutines. This error could be either a syntax error in the FORMAT argument, or the result of specifying a field width which is too small (referred to as an "output conversion error" in Fortran).

  Errors in message construction which result from undefined tokens are *not* considered fatal and STATUS is not set as a result.

- Message Output – If an error occurs when `msgOut`, `msgOutif`, `msgBlank` or `msgBlankif` attempt to output the message, then the STATUS argument will be set to an error value and an error message will be reported (see §3.4). Errors writing the message text to the standard output stream may occur for a number of reasons – the seriousness of such an error is dependent upon where the message was intended to go:

  – Command terminal or VAX/VMS batch log file – In this case the likelihood of being able to inform the user about the error is small. Generally, the application has little chance of continuing successfully under these circumstances.
  – Output file – The message output may have been re-directed to an output file for the duration of a particular application. Under these circumstances, the user can probably be informed about the error and the application may be able to continue.

3 ERR – Error Reporting System

3.1 Overview

Although the Message Reporting System could be used for reporting errors, there are a number of considerations which demand that separate facilities are available for this:

- The inherited status scheme is used by the Message System, and so the MSG output subroutines will not execute if STATUS is set to an error value. Consequently, the Message System cannot be used to report information about an earlier error which has resulted in STATUS being set.
- In a program or package consisting of many levels of subroutines, each routine which has something informative to say about the error should be able to contribute to the information that the user receives. This includes:
  - The subroutine which first detects the error, as this will probably have access to specific information which is hidden from higher level routines.
  - The chain of subroutines between the main program and the routine in which the error originated. Some of these will usually be able to report on the context in which the error occurred, and so add relevant information which is not available to routines at lower levels.

  This can lead to several error reports arising from a single failure.
- It is not always necessary for an error report to reach the user. For example, a high-level subroutine or the main program, may decide that it can handle an error detected at a lower level safely without informing the user. In this case, it is necessary for error reports associated with that error to be discarded, and this can only happen if the output of error messages to the user is deferred.

These considerations have led to the design and implementation of a set of subroutines which form the Error Reporting System. The subroutines have names of the form:

```
ERR_name
```

where `name` indicates what the subroutine does. These subroutines work in conjunction with the Message System and allow error messages to incorporate message tokens.

3.2 Inherited status checking

The recommended method of indicating when errors have occurred within Starlink software is to use an integer status value in each subroutine argument list. This inherited status argument, say STATUS, should always be the last argument and every subroutine should check its value on entry. The principle is as follows:

- The subroutine returns without action if the given value of STATUS is not `SAI__OK`.
- The subroutine leaves STATUS unchanged if it completes successfully.
- The subroutine sets STATUS to an appropriate error value and reports an error message if it fails to complete successfully.

Here is an example of the use of inherited status within a simple subroutine:

```
      SUBROUTINE ROUTN( VALUE, STATUS )

*  Define the SAI__OK global constant.
      INCLUDE 'SAE_PAR'
      INTEGER STATUS

      ...

*  Check the inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

      <application code>

      END
```

If an error occurs within the "application code" of such a subroutine, then STATUS is set to a value which is not `SAI__OK`, an error is reported (see below) and the subroutine aborts.

Note that it is often useful to use a status argument and inherited status checking in subroutines which "cannot fail". This prevents them executing, possibly producing a run-time error, if their arguments contain rubbish after a previous error. Every piece of software that calls such a routine is then saved from making an extra status check. Furthermore, if the routine is later upgraded it may acquire the potential to fail, and so a status argument will subsequently be required. If a status argument is included initially, existing code which calls the routine will not need to be changed (see further discussion of this in §3.17).

3.3 Setting and defining status values

The use of the global constants `SAI__OK` and `SAI__ERROR` for setting status values is recommended in general applications. These global constants may be defined in each subroutine by including the file SAE_PAR at the beginning of the subroutine, prior to the declaration of any subroutine arguments or local variables. When writing subroutine libraries, however, it is useful to have a larger number of globally unique error codes available and to define symbolic constants for these in a separate include file. The naming convention:

```
      fac__ecode
```

should be used for the names of error codes defined in this way; where `fac` is the three-character facility prefix and `ecode` is up to five alphanumeric characters of error code name. *Note the double underscore used in this naming convention.* The include file should be referred to by the name fac_ERR, *e.g.*

```
      INCLUDE 'SGS_ERR'
```

where the facility name is `SGS`, the Starlink Simple Graphics System, in this case. These symbolic constants should be defined at the beginning of every subroutine which requires them, prior to the declaration of any subroutine arguments or local variables.

The purpose of error codes is to enable the status argument to indicate that an error has occurred by having a value which is not equal to `SAI__OK`. By using a set of predefined error codes the calling module is able to test the returned status to distinguish between error conditions which may require different action. It is not generally necessary to define a very large number of error codes which would allow a unique value to be used every time an error report is made. It is sufficient to be able to distinguish the important classes of error which may occur. Examples of existing software can be consulted as a guide in this matter.

The Starlink utility MESSGEN (see SUN/185) should be used on UNIX to generate a set of globally unique error codes for a package. It may be used to create the Fortran include file and/or a C header file defining symbolic names for the error codes, and/or the "facility error message file", which can be used to associate a simple message with each error code (see §3.15). There is an alternative but compatible method of calculating the set of error codes for a package described in Appendix G.

Software from outside a package which defines a set of error codes may use that package's codes to test for specific error conditions arising within that package. However, with the exception of the SAI__ codes, it should *not* assign these values to the status argument. To do so could cause confusion about which package detected the error.

3.4 Reporting errors

The subroutine used to report errors is ERR_REP. It has a calling sequence of the form

```
      CALL ERR_REP( PARAM, TEXT, STATUS )
```

Here, the argument PARAM is the error message name, TEXT is the error message text and STATUS is the inherited status. These arguments are broadly similar to those used in the Message System subroutine MSG_OUT.

The error message name PARAM should be a globally unique identifier for the error report. It is recommended that it has the form:

```
     routn_message
```

in the general case of subroutines within an application, or:

```
     fac_routn_message
```

in the case of routines within a subroutine library. In the former case, `routn` is the name of the application routine from which ERR_REP is being called and `message` is a sequence of characters uniquely identifying the error report within that subroutine. In the latter case, `fac_routn` is the full name of the subroutine from which ERR_REP is being called (see the Starlink Application Programming Standard , SGP/16, for a discussion of the recommended subroutine naming convention), and `message` is a sequence of characters unique within that subroutine. These naming conventions are designed to ensure that each individual error report made within a complete software system has a unique error name associated with it.

Here is a simple example of error reporting where part of the application code of the previous example detects an invalid value of some kind, sets STATUS, reports the error and then aborts:

```
      IF ( <value invalid> ) THEN
         STATUS = SAI__ERROR
         CALL ERR_REP( 'ROUTN_BADV', 'Value is invalid.', STATUS )
         GO TO 999
      END IF

      ...

 999  CONTINUE
      END
```

In the event of an invalid value, the Error System would produce a message like:

```
      !! Value is invalid.
```

Note that when the message is output to the user, the Error System precedes the given text with exclamation marks. For more information on this, see §3.11.

The sequence of three operations:

(1) Set STATUS to an error value.
(2) Report an error.
(3) Abort.

is the standard response to an error condition and should be adopted by all software which uses the Error System.

Note that the behaviour of the STATUS argument in ERR_REP differs somewhat from that in MSG_OUT in that ERR_REP will execute regardless of the input value of STATUS. Although the Starlink convention is for subroutines not to execute if their status argument indicates a previous error, the Error System subroutines obviously cannot behave in this way if their purpose is to report these errors.

On exit from ERR_REP the value of STATUS remains unchanged, with three exceptions:

- If ERR_REP is called with STATUS set to `SAI__OK` – in this case an additional error message to this effect is stacked for output to the user and STATUS is returned set to ERR__BADOK.
- If ERR_REP is unable to output the error message – in this case STATUS is returned set to ERR__OPTER.
- If an internal fault occurs. Currently there are no other faults which will return an error status but the possibility should be allowed for.

3.5 Message tokens in error messages

Message tokens can be used in the error text presented to ERR_REP in the same manner as their use in calls to MSG_OUT, MSG_OUTIF and MSG_LOAD. Here is an example where two values, LOWER and UPPER, are in conflict:

```
*  Check if LOWER and UPPER are in conflict.
      IF ( LOWER .GT. UPPER ) THEN

*     Construct and report the error message.
         STATUS = SAI__ERROR
         CALL MSG_SETI( 'LO', LOWER )
         CALL MSG_SETI( 'UP', UPPER )
         CALL ERR_REP( 'BOUND_ERR',
     :                 'LOWER(^LO) is greater than UPPER(^UP).', STATUS )
         GO TO 999
      END IF
```

If the value of LOWER is 50 and the value of UPPER is 10, then the user might receive a message like:

```
!! LOWER(50) is greater than UPPER(10).
```

After a call to ERR_REP, all message tokens are left undefined.

3.6 When to report an error

In the following example, part of an application makes a series of subroutine calls:

```
      CALL ROUTN1( A, B, STATUS )
      CALL ROUTN2( C, STATUS )
      CALL ROUTN3( T, Z, STATUS )

*  Check the global status.
      IF ( STATUS .NE. SAI__OK ) GO TO 999

      ...

 999  CONTINUE
      END
```

Each of these subroutines uses the inherited status strategy and makes error reports by calling ERR_REP. If an error occurs within any of the subroutines, STATUS will be set to an error value by that routine and inherited status checking by all subsequent routines will cause them not to execute. Thus, it becomes unnecessary to check for an error after each subroutine call, and a single check at the end of a sequence of calls is all that is required to correctly handle any error condition that may arise. Because an error report will already have been made by the subroutine that failed, it is usually sufficient simply to abort if an error arises in a sequence of subroutine calls.

It is important to distinguish the case where a called subroutine sets STATUS and makes its own error report, as above, from the case where STATUS is set explicitly as a result of a directly detected error, as in the previous example. If the error reporting strategy is to function correctly, then responsibility for reporting the error must lie with the routine which modifies the status argument. The golden rule is therefore:

> *If STATUS is explicitly set to an error value, then an accompanying call to ERR_REP* must *be made.*

Unless there are good documented reasons why this cannot be done, subroutines which return a status value and do not make an accompanying error report should be regarded as containing a bug[1].

3.7 The content of error messages

The purpose of an error message is to be informative and it should therefore provide as much relevant information about the context of the error as possible. It must also avoid the danger of being misleading, or of containing too much irrelevant information which might be confusing to a user. Particular care is necessary when reporting errors from within subroutines which might be called by a wide variety of software. Such reports must not make unjustified assumptions about what sort of application might be calling them. For example, in a routine that adds two arrays, the report:

```
!! Error adding two arrays.
```

would be preferable to:

```
!! Error adding two images.
```

if the same routine could be called to add two spectra!

The name of the routine which called ERR_REP to make an error report can often be a vital piece of information when trying to understand what went wrong. However, the error report is intended for the user, not the programmer, and so the name of an obscure internal routine is more likely to confuse than to clarify the nature of the error. A good rule of thumb is to include the names of routines in error reports only if those names also appear in documentation – so that the function they perform can be discovered without delving into the code. An example of this appears in the next section.

[1]For historical reasons there are still some routines in ADAM which set a status value without making an accompanying error report – these are gradually being corrected. If such a routine is used before it has been corrected, then the strategy outlined here is recommended. It is advisable not to complicate new code by attempting to make an error report on behalf of the faulty subroutine. If it is appropriate, please ensure that the relevant support person is made aware of the problem.

3.8 Adding contextual information

Instead of simply aborting when a status value is set by a called subroutine, it is also possible for an application to add further information about the circumstances surrounding the error. In the following example, an application makes several calls to a subroutine which might return an error status value. In each case, it reports a further error message so that it is clear which operation was being performed when the lower-level error occurred:

```
*  Smooth the sky values.
      CALL SMOOTH( NX, NY, SKY, STATUS )
      IF ( STATUS .NE. SAI__OK ) THEN
         CALL ERR_REP( 'SKYOFF_SKY',
     :                'SKYOFF: Failed to smooth sky values.', STATUS )
         GO TO 999
      END IF

*  Smooth the object values.
      CALL SMOOTH( NX, NY, OBJECT, STATUS )
      IF ( STATUS .NE. SAI__OK ) THEN
         CALL ERR_REP( 'SKYOFF_OBJ',
     :                'SKYOFF: Failed to smooth object values.',
     :                STATUS )
         GO TO 999
      END IF

      ...

999   CONTINUE
      END
```

Notice how an additional error report is made in each case, but because the original status value contains information about the precise nature of the error which occurred within the subroutine SMOOTH, it is left unchanged.

If the first call to subroutine SMOOTH were to fail, say because it could not find any valid pixels in the image it was smoothing, then the error message the user would receive might be:

```
!! Image contains no valid pixels to smooth.
!  SKYOFF: Error smoothing sky image.
```

The first part of this message originates from within the subroutine SMOOTH, while the second part qualifies the earlier report, making it clear how the error has arisen. Since SKYOFF is the name of an application known to the user, it has been included in the contextual error message.

This technique can often be very useful in simplifying error diagnosis, but it should not be overdone; the practice of reporting errors at *every* level in a program hierarchy tends to produce a flood of redundant messages. As an example of good practice for a subroutine library, an error report made when an error is first detected, followed by a further contextual error report from the "top-level" routine which the user actually called, normally suffices to produce very helpful error messages.

3.9 Deferred error reporting

The action of the subroutine ERR_REP is to report an error to the Error System but the Error System has the capacity to defer the output of that message to the user. This allows the final delivery of error messages to be controlled within applications software, and this control is achieved using the subroutines ERR_MARK, ERR_RLSE, ERR_FLUSH and ERR_ANNUL. This section describes the function of these subroutines and how they are used.

Subroutine ERR_MARK has the effect of ensuring that all subsequent error messages are deferred by the Error System and stored in an "error table" instead of being delivered immediately to the user. ERR_MARK also starts a new "error context" which has its own table of error messages and message tokens which are independent of those in the previous error context. A return to the previous context can later be made by calling ERR_RLSE. When ERR_RLSE is called, the new error context created by ERR_MARK ceases to exist and any error messages stored in it are transferred to the previous context. Calls to ERR_MARK and ERR_RLSE can be nested if required but should always occur in matching pairs. In this way, no existing error messages can be lost through the deferral mechanism.

The system starts at base-level context (level 1) – at this level, error messages are output to the user immediately. If a call to ERR_RLSE returns the system to base-level context, any messages still stored in the error table will be automatically delivered to the user.

The purpose of deferred error reporting can be illustrated by the following example. Consider a subroutine, say HELPER, which detects an error during execution. The subroutine HELPER reports the error that has occurred, giving as much contextual information about the error as it can. It also returns an error status value, enabling the software that called it to react to the failure appropriately. However, what may be considered an "error" at the level of subroutine HELPER, *e.g.* an "end of file" condition, may be considered by the calling module to be a case which can be handled without informing the user, *e.g.* by simply terminating its input sequence. Thus, although the subroutine HELPER will always report the error condition, it is not always necessary for the associated error message to reach the user. The deferral of error reporting enables application programs to handle such error conditions internally.

Here is a schematic example of what subroutine HELPER might look like:

```
      SUBROUTINE HELPER( LINE, STATUS )

      ...

*  Check if a Fortran I/O error has occurred.
      IF ( IOSTAT .NE. 0 ) THEN

*     Set STATUS and report the error.
         IF ( IOSTAT .LT. 0 ) THEN

*        Report an end-of-file error.
            STATUS = <end-of-file error code>
            CALL ERR_REP( 'HELPER_FIOER',
     :         'Fortran I/O error: end of input file reached', STATUS )
         ELSE

*        Report a Fortran I/O error.
```

```
               STATUS = SAI__ERROR
               CALL ERR_REP( 'HELPER_FIOER',
     :            'Fortran I/O error encountered during data input',
     :            STATUS )
            END IF

*     Abort.
            GO TO 999
         END IF

         ...

 999  CONTINUE
      END
```

Suppose HELPER is called and reports an error, returning with STATUS set. At this point, the error message may, or may not, have been received by the user – this will depend on the environment in which the routine is running, and on whether the software which called HELPER took any action to defer the error report. HELPER itself does not need to take action (indeed it should *not* take action) to ensure delivery of the message to the user; its responsibility ends when it aborts, and responsibility for handling the error condition then passes to the software which called it.

Now suppose that the subroutine HELPED calls HELPER and wishes to defer any messages from HELPER so that it can decide how to handle error conditions itself, rather than troubling the user with spurious messages. It can do this by calling the routine ERR_MARK before it calls HELPER.

The operation of error message deferral can be illustrated by a simple example:

```
      SUBROUTINE HELPED( STATUS )

      ...

*  Create a new error context.
      CALL ERR_MARK

      <any error messages from HELPER are now deferred>

      CALL HELPER( LINE, STATUS )

      ...
```

By calling ERR_MARK before calling HELPER, subroutine HELPED ensures that any error messages reported by HELPER are deferred, *i.e.* held in the error table. HELPED can then handle the error condition itself in one of two ways:

- By calling ERR_ANNUL(STATUS), which "annuls" the error, deleting any deferred error messages in the current context and resetting STATUS to `SAI__OK`. This effectively causes the error condition to be ignored. For instance, it might be used if an "end of file" condition was expected, but was to be ignored and some appropriate action taken instead. (A call to ERR_REP could also be used after ERR_ANNUL to replace the initial error condition with another more appropriate one, although this is not often done.)

- By calling ERR_FLUSH(STATUS), which "flushes out" the error, sending any deferred error messages in the current context to the user and resetting STATUS to `SAI__OK`. This notifies the user that a problem has occurred, but allows the application to continue anyway. For instance, it might be used if a series of files were being read: if one of these files could not be accessed, then the user could be informed of this by calling ERR_FLUSH before going on to process the next file.

Here is the previous example, elaborated to demonstrate the use of ERR_ANNUL. It shows how an "end of file" condition from HELPER might be detected, annulled, and stored by HELPED in a logical variable EOF for later use:

```
*  Initialise end-of-file flag, EOF.
      EOF = .FALSE.

*  Create a new error context.
      CALL ERR_MARK

*  Read line of data.
      CALL HELPER( LINE, STATUS )

*  Trap end-of-file error status and annul any reported error messages
*  for the current error context.
      IF ( STATUS .EQ. <end-of-file error status> ) THEN
         CALL ERR_ANNUL( STATUS )
         EOF = .TRUE.
      END IF

*  Release the current error context.
      CALL ERR_RLSE

*  Abort application on error.
      IF ( STATUS .NE. SAI__OK ) GO TO 999

      ...

999   CONTINUE
      END
```

Note that the routine chooses only to handle "end of file" error conditions; any other error condition will not be annulled and will subsequently cause an abort when STATUS is checked after the call to ERR_RLSE.

Here is an example showing how both ERR_FLUSH and ERR_ANNUL may be used during the process of acquiring a value from the user via a call to the subroutine RDPAR:

```
*  Create a new error context.
      CALL ERR_MARK

*  Loop to get DSCALE parameter value.
      DO WHILE ( .TRUE. )
         CALL RDPAR( 'DSCALE', DSCALE, STATUS )
```

```
*     Check the returned global status.
         IF ( STATUS .EQ. SAI__OK ) THEN

*        Success, so continue with the application.
            GO TO 10
         ELSE IF ( STATUS .EQ. <abort status> ) THEN

*        User wanted to abort, so abort the application.
            CALL ERR_RLSE
            GO TO 999
         ELSE IF ( STATUS .EQ. <null status> ) THEN

*        User entered "null", so annul the error and supply a default.
            CALL ERR_ANNUL( STATUS )
            DSCALE = 1.0
            GO TO 10
         ELSE

*        An error has occurred, so ensure the user knows about it before
*        trying again.
            CALL ERR_FLUSH( STATUS )
            CALL CNPAR( 'DSCALE', STATUS )
         END IF
      END DO

 10   CONTINUE

*  Release the current error context.
      CALL ERR_RLSE

      ...

 999  CONTINUE
      END
```

Note how ERR_FLUSH is used to ensure that any error messages are output to the user before trying again to get a new value. In effect, it passes responsibility for the error condition to the user. This interactive situation is typical of how ERR_FLUSH should be used; it is not needed very often during normal error reporting, and it is certainly *not* required as a regular means of ensuring the delivery of error messages following calls to ERR_REP – this should be left to the Error System itself when returned to the base-level context.

Note that if ERR_FLUSH cannot output the error message to the user, then it will return the error status ERR__OPTER. This allows critical applications to attempt to recover in the event of the failure of the Error System.

Finally, as a safety feature, if ERR_FLUSH is called when no errors have been reported, it outputs the message

```
!! No error to report (improper use of EMS).
```

This is to highlight problems where the inherited status has been set by some item of software, but no accompanying error message has been reported.

3.10 Error table limits

The error table can contain up to 32 error messages, normally reported at different levels within the hierarchy of a structured program. If an attempt is made to defer the reporting of more than 32 error messages, then the last reported error message will be replaced by the message:

```
!! Error message stack overflow (EMS fault).
```

There are up to 256 context levels available in the Error System, the initial (base-level) error context level being 1. The current error context level may be inquired using a call to ERR_LEVEL. If an attempt is made to mark a context level beyond 256, the error message:

```
!! Error context stack overflow (EMS fault).
```

is placed on the error stack at context level 256 and any subsequent error reports will be placed at context level 256. A bug report should be made if either of the "EMS fault" error messages are reported from software.

3.11 Format of delivered messages

When messages are delivered to the user, the Error System prefixes the given text with exclamation marks to call attention to the message and to distinguish between error messages and normal informational messages output using MSG. When a sequence of deferred messages is flushed, the first will be prefixed by '`!! `' and the remainder by '`!  `'.

By default, messages are split so that output lines do not exceed 79 characters – the split is made on word boundaries if possible. The maximum output line size can be altered using tuning parameters (see §4). If a message has to be split for delivery by the Error System, text on continuation lines is indented by three spaces, *e.g.*

```
!! The first line of an error message ...
!     and its continuation onto another line.
!  A second contextual error message.
```

3.12 Routines which perform "cleaning-up" operations

If a subroutine performs "cleaning-up" operations which must execute even if the inherited status has been set, then a different sequence of status checking must usually be performed. The deferral of error messages may also be involved.

Normally, the effect required is that a cleaning-up routine called with its status argument set to `SAI__OK` will behave like any other routine, setting the status value and reporting an error if it fails. However, if the value of status has been set to an error condition because of a previous error, it must still attempt to execute, even if there is a good chance that it will not succeed. In this latter case, an error report is not normally required from the cleaning-up routine.

The following is a typical example:

```
            CALL ALLOC( NBYTES, PNTR, STATUS )

            <application code>

            CALL DEALL( NBYTES, PNTR, STATUS )
```

Here, ALLOC allocates some memory for use by the "application code" and DEALL de-allocates it at the end. The following error conditions may arise:

- DEALL fails – In this case we want to receive an error message from DEALL saying why this happened.
- The application code fails - In this case STATUS will be set to an error value, but DEALL must still execute in order to recover the allocated memory.
- ALLOC fails – In this case STATUS will be set to an error value, and DEALL will attempt to execute, but will also fail because there is no memory to deallocate. In this case, we normally only want to receive an error message from ALLOC.

The solution is to write DEALL so that it saves the value of STATUS on entry and restores it again on exit. To preserve the associated error messages, calls to ERR_MARK and ERR_RLSE are also required. For example:

```
            SUBROUTINE DEALL( NBYTES, PNTR, STATUS )

            ...

      *  Save the initial status value and set a new value for this routine.
            ISTAT = STATUS
            STATUS = SAI__OK

      *  Create a new error context.
            CALL ERR_MARK

            <clean-up code>

      *  If the initial status was bad, then ignore all internal errors.
            IF ( ISTAT .NE. SAI__OK ) THEN
               CALL ERR_ANNUL( STATUS )
               STATUS = ISTAT
            END IF

      *  Release the current error context.
            CALL ERR_RLSE

            END
```

Note how a new error context is used to constrain ERR_ANNUL to annulling only errors arising from the "clean-up code", and not the pre-existing error condition which is to be preserved.

Two routines are provided to "wrap up" these clean-up calls: ERR_BEGIN and ERR_END, which begin and end what is effectively a new error reporting environment. ERR_BEGIN will begin a new error context and return the status value set to SAI__OK. A call to ERR_END will annul the current error context if the previous context contains undelivered error messages. It will then release the current error context. ERR_END returns the status of the last reported error message pending delivery to the user after the current error context has been released. If there are no error messages pending output, then the status is returned set to SAI__OK. This behaviour is exactly that represented by the code in the previous example. Here is the previous example re-written using calls to ERR_BEGIN and ERR_END:

```
      SUBROUTINE DEALL( NBYTES, PNTR, STATUS )

      ...

*  Begin a new error reporting environment.
      CALL ERR_BEGIN( STATUS )

      <clean-up code>

*  End the current error reporting environment.
      CALL ERR_END( STATUS )

      END
```

Like ERR_MARK and ERR_RLSE, ERR_BEGIN and ERR_END should always occur in pairs and can be nested if required. If ERR_BEGIN is called with STATUS set to an error value, then a check is made to determine if there are any error messages pending output at the current error context; if there are not then the status has been set without making an error report. In these cases ERR_BEGIN will make the error report:

```
!! Status set with no error report (improper use of EMS).
```

using the given status value before marking a new error context.

Any code which attempts to execute when the inherited status is set to an error value should be regarded as "cleaning-up".

3.13 Intercepting error messages

It may sometimes be convenient within an application to obtain access to any error messages within the current context via a character variable, instead of the error output stream. The Error System provides subroutine ERR_LOAD to do this; it has the calling sequence:

```
      CALL ERR_LOAD( PARAM, PARLEN, OPSTR, OPLEN, STATUS )
```

The behaviour of ERR_LOAD is the same as ERR_FLUSH except that, instead of delivering deferred error messages from the current error context to the user, the error messages are returned, one by one, through character variables in a series of calls to ERR_LOAD.

On the first call of this routine, the error table for the current context is copied into a holding area, the current error context is annulled and the first message in the holding area is returned.

Thereafter, each time the routine is called, the next message from the holding area is returned. The argument PARAM is the returned message name and PARLEN the length of the message name in PARAM. OPSTR is the returned error message text and OPLEN is the length of the error message in OPSTR.

The status associated with the returned message is returned in STATUS until there are no more messages to return – then STATUS is set to `SAI__OK`, PARAM and OPSTR are set to blanks and PARLEN and OPLEN to 1. As for ERR_FLUSH, a warning message is generated if there are no messages initially. The status returned with the warning message is EMS__NOMSG.

After STATUS has been returned `SAI__OK`, the whole process is repeated for subsequent calls.

The symbolic constants ERR__SZPAR and ERR__SZMSG are provided for declaring the lengths of character variables which are to receive message names and error messages in this way. These constants are defined in the include file ERR_PAR (see §6).

3.14 Protecting tokens

As a general rule, message tokens should be assigned, using calls to the MSG_SET*x* and MSG_FMT*x* routines, immediately prior to the call in which they are to be used. However, this is not always convenient; *e.g.* within an iteration or a block IF statement where the same tokens may be used in one of several potential message reports. Under these circumstances, it is important to protect the values of assigned message tokens when subroutines which may fail are called – when a subroutine fails it must be assumed that it will make an accompanying error report using ERR_REP within the existing error reporting context, thereby annulling any currently defined message tokens. The only sure way of protecting against such behaviour is to bracket the subroutine call which may fail with calls to ERR_MARK and ERR_RLSE. The same precautions are needed when any subroutine is called which may in turn call any of MSG_OUT, MSG_OUTIF, MSG_LOAD, ERR_REP or ERR_LOAD (all of which annul tokens).

It is not good practice to assign message tokens which are to be used in another subroutine.

Here is an example of assigning message tokens outside a block IF statement to be used by ERR_REP and MSG_OUT calls within the IF block. The code is a fragment of a routine for re-scaling a single array to a mean of unity. If the call to the subroutine MEAN fails, any assigned message tokens in the current error reporting context may be annulled; hence the need to bracket this call by calls to ERR_MARK and ERR_RLSE.

```
*  Get the data arrays.
      CALL GETDAT( X, Y, QUAL, NDATA, STATUS )

*  Check the returned status.
      IF ( STATUS .EQ. SAI__OK ) THEN

*     The data have been obtained successfully, assign the token value
*     and inform the user of the number of data obtained.
         CALL MSG_SETI( 'NDATA', NDATA )

         IF ( NDATA .LE. 0 ) THEN

*        No data exist, report an error message and abort.
            STATUS = SAI__ERROR
            CALL ERR_REP( 'NDATA_INVAL',
     :                    'Cannot use this number of data (^NDATA).',
     :                    STATUS )
         ELSE

*        Get the mean of the data.
            CALL ERR_MARK
            CALL MEAN( NDATA, Y, QUAL, MEAN, STATUS )
            CALL ERR_RLSE

*        Check the returned status.
            IF ( STATUS .EQ. SAI__OK ) THEN

*           Deliver the number of data and their mean to the user.
               CALL MSG_SETR( 'MEAN', MEAN )

               IF ( NDATA .EQ. 1 ) THEN
                  CALL MSG_OUT( ' ',
     :               '^NDATA data value (^MEAN) will be used.',
     :               STATUS )
               ELSE
                  CALL MSG_OUT( ' ',
     :               '^NDATA data values with a mean of ^MEAN' //
     :               ' will be used.', STATUS )
               END IF
            ELSE

*           Failed to calculate a mean value for the data (the quality
*           flags were probably all bad). Report an error and abort.
               IF ( NDATA .EQ. 1 ) THEN
                  CALL MSG_SETC( 'VALUE', 'value' )
               ELSE
                  CALL MSG_SETC( 'VALUE', 'values' )
               END IF

               CALL ERR_REP( 'BAD_DATA',
     :            'No mean available for ^NDATA ^VALUE,', //
     :            ' cannot rescale the data.', STATUS )
            END IF
         END IF
      END IF
```

3.15 Reporting Status, Fortran I/O and operating system errors

Some of the lower-level Starlink libraries cannot use ERR (or EMS) to make error reports; furthermore, some items of software may need to perform Fortran I/O operations or calls to operating system routines. Any of these may fail but will not have made error reports through ERR_REP. For this reason it is sometimes useful to convert the given error code into a message which can be displayed as part of a message at a higher level where some context information can be added.

Three subroutines exist to enable a message token to be built from the error code returned under these circumstances. These subroutines are:

```
ERR_FACER( TOKEN, STATUS )
```

where STATUS is a standard Starlink facility status value,

```
ERR_FIOER( TOKEN, IOSTAT )
```

where IOSTAT is a Fortran I/O status code, and

```
ERR_SYSER( TOKEN, SYSTAT )
```

where SYSTAT is a status value returned from an operating system routine.

Each of the above routines will assign the message associated with the given error code to the specified token, appending the message if the token is already defined. The error code argument is never altered by these routines. It is important that the correct routine is called, otherwise the wrong message or, at best, only an error number will be obtained.

ERR_FACER is not likely to be useful for applications programmers because suitable error reports will probably have been made by higher-level facilities called directly by the application – it is really provided for completeness. The other two routines will be more useful.

Here is an example of using ERR_FIOER. It is a section of code that writes a character variable to a formatted sequential file, given the Fortran logical unit of the file:

```
*  Write the character variable STR.
      WRITE( UNIT, '(A)', IOSTAT = IOSTAT ) STR

*  Check the Fortran I/O status.
      IF ( IOSTAT .NE. 0 ) THEN

*     Fortran write error, so set STATUS.
         STATUS = SAI__ERROR

*     Define the I/O status and logical unit message tokens and attempt
*     to obtain the file name.
         CALL ERR_FIOER( 'MESSAGE', IOSTAT )
         CALL MSG_SETI( 'UNIT', UNIT )
         INQUIRE( UNIT, NAME = FNAME, IOSTAT = IOS )

*     Check the returned I/O status from the INQUIRE statement and act.
         IF ( IOS .EQ. 0 ) THEN

*        Define the file name message token.
            CALL MSG_SETC( 'FNAME', FNAME )

*        Report the error.
            CALL ERR_REP( 'PUTSTR_WRERR',
     :                 'Error writing to file ^FNAME on ' //
     :                 'unit ^UNIT: ^MESSAGE', STATUS )
         ELSE
```

```
*        No file name has been found so just report the error.
            CALL ERR_REP( 'PUTSTR_WRERR',
     :                'Error writing to unit ^UNIT: ^MESSAGE', STATUS )
         END IF

         GO TO 999
      END IF

      ...

 999  CONTINUE
      END
```

Here, the name of the file being read is also obtained in order to construct a comprehensive error message, which might be something like:

```
!! Error writing to file BLOGGS.DAT on unit 17: Disk quota exceeded.
```

Note that the I/O status values used in Fortran do not have universally defined meanings except for zero (meaning no error), but by using ERR_FIOER it is still possible to make high quality error reports about Fortran I/O errors in a portable manner.

In a similar way, the subroutine ERR_SYSER may be used to assign an operating system message associated with the system status flag SYSTAT to the named message token. Of course, software that calls operating system routines directly cannot be portable, but ERR_SYSER provides a convenient interface for reporting errors that occur in such routines in a form that can be easily changed if necessary. For example:

```
      IF ( <system error condition> ) THEN

*     Operating system error, so set STATUS.
         STATUS = SAI__ERROR

*     Report the error and abort.
         CALL ERR_SYSER( 'ERRMSG', SYSTAT )
         CALL ERR_REP( 'ROUTN_SYSER', 'System error: ^ERRMSG', STATUS )
         GO TO 999
      END IF

      ...

 999  CONTINUE
      END
```

Fortran I/O and operating system error messages, obtained through calls to ERR_FIOER and ERR_SYSER respectively, will differ depending upon which operating system (or even flavour of operating system) an application is run on.

Because of the necessary generality of these messages (and those from ERR_FACER), many will appear rather vague and unhelpful without additional contextual information. This is particularly true of UNIX implementations. It is very important to provide additional contextual information when using these routines in order to avoid obfuscating rather than clarifying the nature of an error. This can be done either as part of the error message which includes the message token set by ERR_FACER, ERR_FIOER or ERR_SYSER, or by making a further error report. The examples in this section provide a good illustration of how this can be done.

3.16 Incorporating foreign routines

Sometimes "foreign" subroutines must be called which do not use the Starlink error status conventions (*e.g.* because they must adhere to some standard interface definition like GKS). Unless they are unusually robust, such routines must normally be prevented from executing under error conditions, either by performing a status check immediately beforehand, or by enclosing them within an appropriate IF...END IF block. Depending on the form of error indication that such foreign routines use, it may also be necessary to check afterwards whether they have succeeded or not. If such a routine fails, then for compatibility with other Starlink software a status value should be set and an error report made on its behalf.

For example, the following code makes a GKS inquiry and checks the success of that inquiry:

```
      IF ( STATUS .EQ. SAI__OK ) THEN

*     Inquire the GKS workstation colour facilities available.
         CALL GQCF( WTYPE, ERRIND, NCOLI, COLA, NPCI )

*     Check if a GKS error has occurred.
         IF ( ERRIND .NE. GKS__OK ) THEN

*        An error has occurred, so report it and abort.
            STATUS = SAI__ERROR
            CALL MSG_SETI( 'ERRIND', ERRIND )
            CALL ERR_REP( 'ROUTN_GQCFERR',
     :              'Error no. ^ERRIND occurred in GKS routine GQCF ' //
                    '(enquire workstation colour facilities).',
     :              STATUS )
            GO TO 999
         END IF
      END IF

      ...

 999  CONTINUE
      END
```

In some cases, it may be possible to obtain a textual error message from the error flag, by means of a suitable inquiry routine, which could be used as the basis of the error report.

It will be obvious from this example how convenient the inherited error status strategy is, and how much extra work is involved in obtaining the same degree of robustness and quality of error reporting from routines which do not use it. It is worth bearing this in mind if you are involved in importing foreign subroutine libraries for use with Starlink software: the provision of a few simple routines for automating error reporting, or an extra layer of subroutine calls where inherited status checking and error reporting can be performed, can make the final product vastly easier to use. Starlink staff will be pleased to offer advice on this matter if consulted.

3.17 Converting existing subroutine libraries

When converting existing subroutine libraries to use the inherited status conventions and the Error System, it is conceivable that an existing subroutine which does not have a status argument will acquire the potential to fail and report an error, either from within itself or from packages layered beneath it. Ideally, the argument list of the subroutine should be changed to include a status argument. However, it may be inconvenient to modify the argument list of a commonly used subroutine (*i.e.* because of the amount of existing code which would have to be changed), and so an alternative method is needed to determine if status has been set during the call so that the appropriate action can be taken by the caller. The subroutine ERR_STAT is provided for recovering the last reported status value under these conditions. Here is an example of the use of ERR_STAT, called from a subroutine which follows the error reporting conventions:

```
*  Call subroutine NOSTAT.
      CALL NOSTAT( GIVARG, RETARG )
      CALL ERR_STAT( STATUS )
```

Here, the calls to NOSTAT and ERR_STAT are equivalent to one subroutine call with a status argument. The use of ERR_STAT to return the current error status relies upon the use of ERR_REP to report errors by the conventions described in this document. In particular, foreign packages *must* be incorporated in the recommended wayas described in §3.16 for ERR_STAT to be reliable.

Finally, it is emphasised that ERR_STAT is *only* for use where there is no other choice than to use this mechanism to determine the last reported error status.

4 Tuning

Some aspects of MSG and ERR systems can be tuned to the user's requirements. Tuning of the two systems is done separately as potentially each could be using entirely different devices. (In practice they share the same device and use EMS at lower levels, so tuning ERR can affect MSG and vice versa.)

Tuning is performed by calling the subroutines MSG_TUNE and/or ERR_TUNE, giving a keyword to specify the parameter to be set, and a value. See the descriptions of MSG_TUNE and ERR_TUNE for details of the parameters and values available and for some further notes.

When the tuning subroutines are called, the given value may be overridden by setting an environment variable. The name of the environment variable is constructed by prefixing the string 'MSG_' or 'ERR_' to the tuning parameter name. An attempt is made to interpret the translation of the environment variable as an integer in the required range for the particular parameter.

For example, if the program contains:

```
CALL MSG_TUNE( 'SZOUT', 79, STATUS )
```

and environment variable MSG_SZOUT is set to 0, no line wrapping occurs on output.

If the environment variable is set but the value is invalid, an error message is reported.

If a tuning subroutine is called with the tuning parameter name set to 'ENVIRONMENT', the given value is ignored but an attempt is made to get values for all possible tuning parameters of the subsystem (MSG or ERR) from their associated environment variable.

5 The C Interface

A preliminary C interface is provided for trial purposes. It may be subject to change in the light of experience.

The interface obeys the rules defined in (PRO)LUN/10.

Briefly, the function name is generated from the Fortran subroutine name by forcing the name to lower case apart from the first character following any underscores, which is forced to upper case. Underscores are then removed.

For example: The C interface function for 'MSG_OUT' is 'msgOut'.

Arguments are provided in the same order as for the Fortran routine with the exception that CHARACTER arrays and returned CHARACTER strings have an additional argument of type `int` (passed by value) immediately following them to specify a maximum length for the output string(s) including the terminating null for which space must be allowed.

There is a fixed relationship between the type of the Fortran argument and the type of the argument supplied to the C function – it is as follows:

Fortran type	C type
INTEGER	int
REAL	float
REAL*8	double
DOUBLE PRECISION	double
LOGICAL	int
CHARACTER	char
FUNCTION	*type* (**name*)()
SUBROUTINE	void (**name*)()

Apart from any argument named 'status', given-only scalar arguments (not including character strings) are passed by value. All others are passed by pointer.

Arrays must be passed with the elements stored in the order required by Fortran.

All necessary constants and function prototypes can be defined by:

```
#include "mers.h"
```

The header file `mers.h` is contained in directory `/star/include`.

6 Compiling and Linking with MSG and ERR

There are five Fortran include files available for use with the Message and Error System: SAE_PAR, MSG_PAR, MSG_ERR, ERR_PAR and ERR_ERR. (See Appendix A for details of the symbolic constants which they define).

The Starlink convention is that the name in upper case with no path or extension is specified when including these files within Fortran code, *e.g.*

```
*  Global Constants:
      INCLUDE 'SAE_PAR'
      INCLUDE 'MSG_PAR'
```

Equivalent header files are provided for use in C code which is calling MSG or ERR – all the required header files, including the function prototypes, may be included by including the file **mers.h**.

The syntax

```
#include "sae_par.h"
#include "mers.h"
```

should be used within the C code.

Assuming that the software has been installed in the standard way and **/star/bin** has been added to the environment variable **PATH**, soft links with these upper-case names pointing to the required file are set up in the user's working directory by the the commands:

```
% star_dev
% err_dev
```

The ADAM version of MSG/ERR is included automatically when programs are linked using the ADAM application linking commands, **alink** *etc*. These will handle either Fortran or C code.

To compile and link a non-ADAM program with the stand-alone version of MSG/ERR, the command line would be, *e.g.*

```
% f77 -o program program.f -L/star/lib `err_link`
```

On platforms with shareable libraries, `-L/star/lib` might be replaced by `-L/star/share`.

If it is necessary to link explicitly with the ADAM version of MSG/ERR and any libraries which it uses (*e.g.* to produce a shareable library), the script **err_link_adam** is available in **/star/bin**. The link command might be:

```
% ld -shared -o libmypkg.so.1.0 -L/star/share -lmypkg `err_link_adam`
```

The compilation of C code should be used with the compiler flag **-I/star/include**. For example the command to compile a C program might be:

```
% cc -c -I/star/include program.c
```

Note that the command used to invoke the C compiler varies from one UNIX implementation to another (indeed, there may be more than one C compiler available on the same machine) – you should therefore check this with your Site Manager before proceeding.

Because the MERS C interface calls Fortran routines it is also necessary to explicitly link with the required Fortran libraries, *e.g.*

```
% cc program.o -L/star/lib ‘err_link‘ -lF77 -lm -o program.out
```

The naming and number of the Fortran libraries differs between UNIX machines and so it is advisable to check in the relevant Fortran documentation for further details.

7 References

Note: Only the first author is listed here.

Lawden, M.D.	SG/4 — ADAM – The Starlink Software Environment.
Rees, P.C.T.	SSN/4 — EMS – Error Message Service.
Chipperfield, A.J.	SUN/185 — MESSGEN – Starlink Facility Error Message Generation.
Wallace, P.T.	SGP/16 — Starlink Application Programming Standard.
Charles A.C.	SUN/40 — CHR – Character Handling Routines.

A Include Files

The symbolic constants defined by the five include files used with MSG and ERR are listed below. The way in which these files are included within Fortran code is described in the section on compiling and linking §6.

SAE_PAR Defines the non-specific status codes for Starlink.

- **SAI__ERROR** – Error encountered.
- **SAI__OK** – No error.
- **SAI__WARN** – Warning.

MSG_PAR Defines the public Message System constants.

- **MSG__NORM** – Normal conditional message output level.
- **MSG__QUIET** – Quiet (few messages) conditional message output level.
- **MSG__SZMSG** – Maximum length of message text.
- **MSG__VERB** – Verbose (abundant messages) conditional message output level.
- **MSG__DEBUG** – Debug (many many messages) conditional message output level.

MSG_ERR Defines the Message Reporting System errors.

- **MSG__BDPAR** – Invalid tuning parameter name (improper use of MSG_TUNE).
- **MSG__BTUNE** – Bad tuning value (improper use of MSG_TUNE).
- **MSG__INVIF** – Invalid conditional message filter value.
- **MSG__BDENV** – Bad MSG environment variable value.
- **MSG__OPTER** – Error encountered during message output.
- **MSG__SYNER** – Error encountered during message synchronisation.

ERR_PAR Defines the public Error System constants.

- **ERR__SZMSG** – Maximum length of error message text.
- **ERR__SZPAR** – Maximum length of error message name.

ERR_ERR Defines the Error Reporting System errors.

- **ERR__BADOK** – Status set to `SAI__OK` in call to ERR_REP (improper use of ERR_REP).
- **ERR__BDPAR** – Invalid tuning parameter name (improper use of ERR_TUNE).
- **ERR__BTUNE** – Bad tuning value (improper use of ERR_TUNE).
- **ERR__BDENV** – Bad ERR environment variable value.
- **ERR__OPTER** – Error encountered during message output.
- **ERR__UNSET** – Status not set in call to ERR_REP (improper use of ERR_REP).

B Subroutine List

B.1 Message System subroutines

MSG_BELL (STATUS)
Deliver an ASCII BEL character.

MSG_BLANK (STATUS)
Output a blank line.

MSG_BLANKIF (FILTER, STATUS)
Conditionally delivers a blank line.

MSG_FMT*x* (TOKEN, FORMAT, VALUE)
Assign a value to a message token (formatted).

MSG_IFLEV (FILTER)
Return the current filter level for conditional message output.

MSG_IFSET (FILTER, STATUS)
Set the filter level for conditional message output.

MSG_LOAD (PARAM, TEXT, OPSTR, OPLEN, STATUS)
Expand and return a message.

MSG_OUT (PARAM, TEXT, STATUS)
Output a message.

MSG_OUTIF (FILTER, PARAM, TEXT, STATUS)
Conditionally deliver the text of a message to the user.

MSG_RENEW
Renew any annulled message tokens in the current context.

MSG_SET*x* (TOKEN, VALUE)
Assign a value to a message token (concise).

MSG_TUNE (PARAM, VALUE, STATUS)
Set an MSG tuning parameter.

B.2 Error System subroutines

ERR_ANNUL (STATUS)
Annul the contents of the current error context.

ERR_BEGIN (STATUS)
Begin a new error reporting environment.

ERR_END (STATUS)
End the current error reporting environment.

ERR_FACER (TOKEN, IOSTAT)
Assign the message associated with a Starlink status to a token.

ERR_FIOER (TOKEN, IOSTAT)
Assign the message associated with a Fortran I/O error to a token.

ERR_FLBEL (STATUS)
Deliver an ASCII BEL and flush the current error context.

ERR_FLUSH (STATUS)
Flush the current error context.

ERR_LEVEL (LEVEL)
Inquire the current error context level.

ERR_LOAD (PARAM, PARLEN, OPSTR, OPLEN, STATUS)
Return error messages from the current error context.

ERR_MARK
Mark (start) a new error context.

ERR_REP (PARAM, TEXT, STATUS)
Report an error message.

ERR_RLSE
Release (end) the current error context.

ERR_STAT (STATUS)
Inquire the last reported error status.

ERR_SYSER (TOKEN, SYSTAT)
Assign the message associated with an operating system error to a token.

ERR_TUNE (PARAM, VALUE, STATUS)
Set an ERR tuning parameter.

B.3 ADAM-special user subroutines

These are only for use in ADAM applications.

MSG_IFGET (PNAME, STATUS)
Get the MSG filter level from the ADAM parameter system.

MSG_SYNC (STATUS)
Synchronise message output via the user interface.

B.4 ADAM-special system subroutines

These are only for use by the ADAM system. They are described in SSN/4 but listed here for completeness.

ERR_CLEAR (STATUS)
Return the error table to the default context and flush its contents.

ERR_START
Initialise the Error Reporting System.

ERR_STOP
Close the Error Reporting System.

C C Interface Function Prototypes

Where *T* is one of d, i, l, r, and *TYPE* is the corresponding C type, the function prototypes for the C language interface are:

void errAnnul(int ∗status);
Annul the contents of the current error context.

void errBegin(int ∗status);
Begin a new error reporting environment.

void errClear(int ∗status);
Return the error table to the default context and flush its contents.

void errEnd(int ∗status);
End the current error reporting environment.

void errFacer(const char ∗token, int ∗status);
Assign the message associated with a Starlink status to a token.

void errFioer(const char ∗token, int iostat);
Assign the message associated with a Fortran I/O error to a token.

void errFlbel(int ∗status);
Deliver an ASCII BEL and flush the current error context.

void errFlush(int ∗status);
Flush the current error context.

void errLevel(int ∗level);
Inquire the current error context level.

void errLoad
(char ∗param, int param_length, int ∗parlen,
char ∗opstr, int opstr_length, int ∗oplen, int ∗status);
Return error messages from the current error context.

void errMark(void);
Mark (start) a new error context.

void errRep(const char ∗param, const char ∗text, int ∗status);
Report an error message.

void errRlse(void);
Release (end) the current error context.

void errStart(void);
Initialise the Error Reporting System.

void errStat(int ∗status);
Inquire the last reported error status.

void errStop(int ∗status);
Close the Error Reporting System.

void errTune(const char ∗param, int value, int ∗status);
Set an ERR tuning parameter.

void msgBell(int ∗status);
Deliver an ASCII BEL character.

void msgBlank(int ∗status);
Output a blank line.

void msgBlankif(int filter, int ∗status);
Conditionally deliver a blank line.

void msgFmtc(const char ∗token, const char ∗format, const char ∗cvalue);
Assign a CHARACTER value to a message token (formatted).

void msgFmt*T*(const char ∗token, const char ∗format, *TYPE* value);
Assign a value to a message token (formatted).

void msgIfget(const char ∗pname, int ∗status);
Get the MSG filter level from the ADAM parameter system.

void msgIflev(int ∗filter);
Return the current filter level for conditional message output.

void msgIfset(int filter, int ∗status);
Set the filter level for conditional message output.

void msgLoad
(const char ∗param, const char ∗text,
char ∗opstr, int opstr_length, int ∗oplen, int ∗status);
Expand and return a message.

void msgOut(const char ∗param, const char ∗text, int ∗status);
Output a message.

void msgOutif(int prior, const char ∗param, const char ∗text, int ∗status);
Conditionally deliver the text of a message to the user.

void msgRenew(void);
Renew any annulled message tokens in the current context.

void msgSetc(const char ∗token, const char ∗cvalue);
Assign a CHARACTER value to a message token (concise).

void msgSet*T*(const char ∗token, *TYPE* value);
Assign a value to a message token (concise).

void msgSync(int ∗status);
Synchronise message output via the user interface.

void errSyser(const char ∗token, int systat);
Assign the message associated with an operating system error to a token.

void msgTune(const char ∗param, int value, int ∗status);
Set an MSG tuning parameter.

D Subroutine Specifications

D.1 Message System subroutines

MSG_BELL
Deliver an ASCII BEL character

Description:

A bell character and a new line is delivered to the user. If the user interface in use supports the ASCII BEL character, this routine will ring a bell and print a new line on the terminal.

Invocation:

```
CALL MSG_BELL( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status.

MSG_BLANK
Output a blank line

Description:

A blank line is output unless the conditional output filter is set to MSG__QUIET. If STATUS is not set to SAI__OK on entry, no action is taken. If an output error occurs, an error report is made and STATUS returned set to MSG__OPTER.

Invocation:

```
CALL MSG_BLANK( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status.

MSG_BLANKIF
Conditionally output a blank line

Description:

Depending upon the given value of the given message priority and the message filtering level set using msgIfset, a blank line is either output to the user or discarded. If the status argument is not set to SAI__OK on entry, no action is taken. If an output error occurs, an error report is made and the status argument returned set to MSG__OPTER.

Invocation:

```
CALL MSG_BLANK( PRIOR, STATUS )
```

Arguments:

PRIOR = INTEGER (Given)

STATUS = INTEGER (Given and Returned)

The global status.

MSG_FMTx
Assign a value to a message token (formatted)

Description:

A given value is encoded using the supplied Fortran 77 format field and the result assigned to the named message token. If the token is already defined, the result is appended to the existing token value. The given value may be one of the following Fortran 77 data types and there is one routine provided for each data type:

Subroutine	*Fortran 77 Type*
MSG_FMTD	DOUBLE PRECISION
MSG_FMTR	REAL
MSG_FMTI	INTEGER
MSG_FMTL	LOGICAL
MSG_FMTC	CHARACTER

If these subroutines fail, it will usually be apparent in any messages which refer to this token.

Invocation:

```
CALL MSG_FMTx( TOKEN, FORMAT, VALUE )
```

Arguments:

TOKEN = CHARACTER ∗ (∗) (Given)
The message token name.

FORMAT = CHARACTER ∗ (∗) (Given)
The Fortran 77 FORMAT field used to encode the supplied value.

VALUE = Fortran 77 type (Given)
The value to be assigned to the message token.

System-specific :

The precise effect of failures will depend upon the computer system being used.

MSG_IFLEV
Return the current filter level for conditional message output

Description:

The value of the current filtering level set for conditional message output is returned. The collating sequence:

MSG__QUIET < MSG__NORM < MSG__VERB

may be assumed.

Invocation:

```
CALL MSG_IFLEV( FILTER )
```

Arguments:

FILTER = INTEGER (Returned)

The current message filtering level.

MSG_IFSET
Set the filter level for conditional message output

Description:

The value of the message filtering level is set using the given filtering value. If no such level exists, then an error is reported, STATUS returned set to MSG__IFINV and the current filtering level remains unchanged.

Invocation:

```
CALL MSG_IFSET( FILTER, STATUS )
```

Arguments:

FILTER = INTEGER (Given)

The filtering level. This may be one of three levels defined in the MSG_PAR include file:

- MSG__QUIET = quiet mode;
- MSG__NORM = normal mode;
- MSG__VERB = verbose mode.
- MSG__DEBUG = debugging mode

STATUS = INTEGER (Given and Returned)

The global status.

MSG_LOAD
Expand and return a message

Description:

Any tokens in the supplied message are expanded and the result is returned in the character variable supplied. If the expanded message is longer than the length of the supplied character variable, the message is terminated with an ellipsis (*i.e.* "...") but no error results.

If STATUS is not set to SAI__OK on entry, no action is taken except to annul any existing message tokens.

Invocation:

```
CALL MSG_LOAD( PARAM, TEXT, OPSTR, OPLEN, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
The message name.

TEXT = CHARACTER * (*) (Given)
The raw message text.

OPSTR = CHARACTER * (*) (Returned)
The expanded message text.

OPLEN = INTEGER (Returned)
The length of the expanded message.

STATUS = INTEGER (Given and Returned)
The global status.

MSG_OUT
Output a message

Description:

Any tokens in the supplied message are expanded and the result is output to the user unless the conditional output filter is set to MSG__QUIET. All existing message tokens are then annulled. If the expanded message exceeds the maximum allowed size, it will be terminated by an ellipsis (*i.e.* "...") but no error results.

If STATUS is not set to SAI__OK on entry, no action is taken except to annul existing message tokens.

If an output error occurs, an error is reported and STATUS is returned set to MSG__OPTER.

A call to MSG_OUT is equivalent to a call to MSG_OUTIF with the message output priority set to MSG__NORM.

Invocation:

```
CALL MSG_OUT( PARAM, TEXT, STATUS )
```

Arguments:

PARAM = CHARACTER ∗ (∗) (Given)
The message name.

TEXT = CHARACTER ∗ (∗) (Given)
The message text.

STATUS = INTEGER (Given and Returned)
The global status.

MSG_OUTIF
Conditionally deliver the text of a message to the user

Description:

If the given message priority, PRIOR, and the current message filtering level (set by MSG_IFSET or MSG_IFGET) indicate that the message should be delivered, the message text is expanded and output to the user. If not, the message is discarded. In either case, any existing message tokens are then annulled. If the expanded message exceeds the maximum allowed size, it will be terminated by an ellipsis (*i.e.* "...") but no error results.

If STATUS is not set to SAI__OK on entry, no action is taken except to annul existing message tokens.

If an output error occurs, an error is reported and STATUS returned set to MSG__OPTER.

Invocation:

```
CALL MSG_OUTIF( PRIOR, PARAM, TEXT, STATUS )
```

Arguments:

PRIOR = INTEGER (Given)

Message output priority. This may be one of three values defined in the MSG_PAR include file:

- MSG__QUIET = Always output the message, regardless of the output filter setting;
- MSG__NORM = Output the message unless the current output filter is set to MSG__QUIET;
- MSG__VERB = Do not output the message unless the current output filter is set to MSG__VERB.
- MSG__DEBUG = Do not output the message unless the current output filter is set to MSG__DEBUG.

If any other value is given, STATUS is set to MSG__INVIF and an error report made; no further action will be taken.

PARAM = CHARACTER ∗ (∗) (Given)

The message name.

TEXT = CHARACTER ∗ (∗) (Given)

The message text.

STATUS = INTEGER (Given and Returned)

The global status.

MSG_RENEW
Renew any annulled message tokens in the current context

Description:

Any message tokens which have been annulled (by a call to MSG_OUT, ERR_REP, *etc.*) are renewed unless any new token value has been defined (using MSG_SET*x*, MSG_FMT*x* *etc.*) since the previous tokens were annulled.

The intended use of MSG_RENEW is to allow a set of message token values to be used repeatedly in a sequence of messages without having to re-define them each time.

Invocation:

```
CALL MSG_RENEW
```

MSG_SETx
Assign a value to a message token (concise)

Description:

A given value is encoded using a concise format and the result assigned to the named message token. If the token is already defined, the result is appended to the existing token value. The given value may be one of the following Fortran 77 data types and there is one routine provided for each data type:

Subroutine	*Fortran 77 Type*
MSG_SETD	DOUBLE PRECISION
MSG_SETR	REAL
MSG_SETI	INTEGER
MSG_SETL	LOGICAL
MSG_SETC	CHARACTER

If these subroutines fail, it will usually be apparent in any messages which refer to this token.

Invocation:

```
CALL MSG_SETx( TOKEN, CVALUE )
```

Arguments:

TOKEN = CHARACTER $*$ ($*$) (Given)
The message token name.

VALUE = Fortran 77 type (Given)
The value to be assigned to the message token.

System-specific :

The precise effect of failures will depend upon the computer system being used.

MSG_TUNE
Set an MSG tuning parameter

Description:

The value of the MSG tuning parameter is set appropriately, according to the value given. MSG_TUNE may be called multiple times for the same parameter.

The given value can be overridden by setting an environment variable, MSG_*PARAM* (where *PARAM* is the tuning parameter name in upper case), at run time.

Invocation:

```
CALL MSG_TUNE( PARAM, VALUE, STATUS )
```

Arguments:

PARAM = CHARACTER$*$($*$) (Given)
The tuning parameter to be set (case insensitive).

VALUE = INTEGER (Given)
The desired value (see Notes).

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

1. The following values of PARAM may be used:

- 'FILTER' Specifies the required MSG conditional message reporting level. VALUE may be 1, 2 or 3, corresponding with quiet, normal (the default) and verbose levels respectively.
- 'SZOUT' Specifies a maximum line length to be used in the line wrapping process. By default the message output by MSG is split into chunks of no more than the maximum line length, and each chunk is written on a new line. The split is made at word boundaries if possible. The default maximum line length is 79 characters.

  If VALUE is set to 0, no wrapping will occur. If it is set greater than 0, it specifies the maximum output line length.
- 'STREAM' Specifies whether or not MSG should treat its output unintelligently as a stream of characters. If VALUE is set to 0 (the default) all non-printing characters are replaced by blanks, and line wrapping occurs (subject to SZOUT). If VALUE is set to 1, no cleaning or line wrapping occurs.
- 'ENVIRONMENT' This is not a true tuning parameter name but causes the environment variables associated with all the true tuning parameters to be used if set. If the environment variable is not set, the tuning parameter is not altered. The VALUE argument is not used.

2. The tuning parameters for MSG and ERR operate partially at the EMS level and may conflict in their requirements of EMS.

3. The use of SZOUT and STREAM may be affected by the message delivery system in use. For example there may be a limit on the the size of a line output by a Fortran WRITE and automatic line wrapping may occur. In particular, a NULL character will terminate a message delivered by the ADAM message system.

D.2 Error System subroutines

ERR_ANNUL
Annul the contents of the current error context

Description:

Any error messages pending output in the current error context are annulled, *i.e.* deleted. The values of any existing message tokens become undefined and STATUS is reset to SAI__OK.

Invocation:

```
CALL ERR_ANNUL( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The global status: it is set to SAI__OK on return.

ERR_BEGIN
Create a new error reporting environment

Description:

Begin a new error reporting environment by marking a new error reporting context and then resetting the status argument to SAI__OK. If ERR_BEGIN is called with STATUS set to an error value, a check is made to determine if there are any messages pending output in the current context: if there are none, an error report to this effect is made on behalf of the calling application.

Invocation:

```
CALL ERR_BEGIN( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status.

ERR_END
End the current error reporting environment

Description:

Check if any error messages are pending output in the previous error reporting context. If there are, the current context is annulled and then released; if not, the current context is just released. The last reported status value is returned on exit.

Invocation:

```
CALL ERR_END( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The global status.

ERR_FACER
Assign a Starlink facility status message to a token

Description:

The text of the error message associated with a Starlink facility status value, STATUS, is assigned to the named message token. If the token is already defined, the message is appended to the existing token value.

Invocation:

```
CALL ERR_FACER( TOKEN, STATUS )
```

Arguments:

TOKEN = CHARACTER $*$ ($*$) (Given)
The message token name.

STATUS = INTEGER (Given)
The Starlink status value.

System-specific :

The messages generated using this facility may differ slightly depending on the computer system upon which the library is implemented.

ERR_FIOER
Assign a Fortran I/O error message to a token

Description:

The text of the error message associated with the Fortran I/O status value, IOSTAT, is assigned to the named message token. If the token is already defined, the message is appended to the existing token value.

Invocation:

```
CALL ERR_FIOER( TOKEN, IOSTAT )
```

Arguments:

TOKEN = CHARACTER * (*) (Given)
The message token name.

IOSTAT = INTEGER (Given)
The Fortran I/O status value.

System-specific :

The messages generated using this facility will depend on the computer system upon which the library is implemented.

ERR_FLBEL
Deliver an ASCII BEL and flush the current error context

Description:

An ASCII BEL character is delivered to the user and then all pending error messages in the current error context are delivered using a call to ERR_FLUSH. On successful completion, the error context is annulled and STATUS reset to SAI__OK.

If no error messages have been reported in the current error context, a warning message is generated but no error results.

If an error occurs during output of the error messages, the error context is not annulled and STATUS is returned set to ERR__OPTER.

Invocation:

```
CALL ERR_FLBEL( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The global status: it is set to SAI__OK on return if the error message output is successful; if not, it is set to ERR__OPTER.

ERR_FLUSH
Flush the current error context

Description:

Ensure that all pending error messages in the current error context have been output to the user. On successful completion, the error context is annulled and STATUS reset to SAI__OK

If no error messages have been reported in the current error context, a warning message is generated but no error results.

If an error occurs during output of the error messages, the error context is not annulled and STATUS is returned set to ERR__OPTER.

Invocation:

```
CALL ERR_FLUSH( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The global status: it is set to SAI__OK on return if the error message output is successful; if not, it is set to ERR__OPTER.

ERR_LEVEL
Inquire the current error context level

Description:

Return the number of context markers set in the error message table.

Invocation:

```
CALL ERR_LEVEL( LEVEL )
```

Arguments:

LEVEL = INTEGER (Returned)

The error context level: all values greater than one indicate the deferral of reported error messages.

ERR_LOAD
Return error messages from the current error context

Description:

On the first call of this routine, the error table for the current error context is copied into a holding area, the current context is annulled and the first message in the holding area is returned. Thereafter, each time the routine is called, the next message from the holding area is returned. The argument PARAM is the returned message name and PARLEN the length of the message name. OPSTR is the returned error message text and OPLEN is the length of the error message. If the message text is longer than the declared length of OPSTR, then the message is truncated with an ellipsis, *i.e.* "...", but no error results.

The status associated with the returned message is returned in STATUS until there are no more messages to return – then STATUS is set to SAI__OK, PARAM and OPSTR are set to blanks and PARLEN and OPLEN to 1. If there are no messages pending on the first call, a warning message is generated and returned with STATUS set to EMS__NOMSG.

After STATUS has been returned SAI__OK, the whole process is repeated for subsequent calls.

Invocation:

```
ERR_LOAD( PARAM, PARLEN, OPSTR, OPLEN, STATUS )
```

Arguments:

PARAM = CHARACTER $*$ ($*$) (Returned)
The error message name.

PARLEN = INTEGER (Returned)
The length of the error message name.

OPSTR = CHARACTER $*$ ($*$) (Returned)
The error message – or blank if there are no more messages.

OPLEN = INTEGER (Returned)
The length of the error message.

STATUS = INTEGER (Returned)
The status associated with the returned error message: it is set to SAI__OK when there are no more messages.

ERR_MARK
Mark (start) a new error context

Description:

Begin a new error reporting context. Delivery of subsequently reported error messages is deferred and the messages held in the error table. Calls to ERR_ANNUL, ERR_FLUSH and ERR_LOAD will only flush or annul the contents of the error table within this new context.

Invocation:

```
CALL ERR_MARK
```

ERR_REP
Report an error message

Description:

Report an error message. According to the error context, the error message is either sent to the user or retained in the error table. The latter case allows the application to take further action before deciding if the user should receive the message. On exit the values associated with any existing message tokens are left undefined. On successful completion, STATUS is returned unchanged.

If STATUS is set to SAI__OK on entry, an error report to this effect is made on behalf of the application and STATUS is returned set to ERR__BADOK; the given message is still reported and has status ERR__UNSET associated with it.

If an output error occurs, STATUS is returned set to ERR__OPTER.

Invocation:

```
CALL ERR_REP( PARAM, TEXT, STATUS )
```

Arguments:

PARAM = CHARACTER $*$ ($*$) (Given)
The error message name.

TEXT = CHARACTER $*$ ($*$) (Given)
The error message text.

STATUS = INTEGER (Given and Returned)
The global status: it is left unchanged on successful completion, or is set an appropriate error value if an internal error has occurred.

ERR_RLSE
Release (end) the current error context

Description:

Release a “mark” in the error message table, returning the Error Reporting System to the previous error context. Any error messages pending output will be passed to this previous context, *not* annulled. When the context level reaches the base level, all pending messages will be delivered to the user.

Invocation:

```
CALL ERR_RLSE
```

ERR_STAT
Inquire the last reported error status

Description:

The current error context is checked for any error messages reported since the context was created or last annulled. If none exist, STATUS is returned set to SAI__OK. If there are any such messages, STATUS is returned set to the status value associated with the last reported error message.

Invocation:

```
CALL ERR_STAT( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The last reported error status within the current error context; if none exist, STATUS is returned set to SAI__OK.

ERR_SYSER
Assign an operating system error message to a token

Description:

The text of the error message associated with the operating system status value, SYSTAT, is assigned to the named message token. If the token is already defined, the message is appended to the existing token value.

Invocation:

```
CALL ERR_SYSER( TOKEN, SYSTAT )
```

Arguments:

TOKEN = CHARACTER ∗ (∗) (Given)
The message token name.

SYSTAT = INTEGER (Given)
The operating system status value.

System-specific :

The messages generated using this facility will depend on the computer system upon which the library is implemented.

ERR_TUNE
Set an ERR tuning parameter

Description:

The value of the ERR tuning parameter is set appropriately, according to the value given. ERR_TUNE may be called multiple times for the same parameter.

The given value can be overridden by setting an environment variable, ERR_*PARAM* (where *PARAM* is the tuning parameter name in upper case), at run time.

The routine will attempt to execute regardless of the given value of STATUS. If the given value is not SAI__OK, then it is left unchanged, even if the routine fails to complete. If the STATUS is SAI__OK on entry and the routine fails to complete, STATUS will be set and an error report made.

Invocation:

```
CALL ERR_TUNE( PARAM, VALUE, STATUS )
```

Arguments:

PARAM = CHARACTER*(*) (Given)
The tuning parameter to be set (case insensitive).

VALUE = INTEGER (Given)
The desired value (see Notes).

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

1. The following values of PARAM may be used:

- ’SZOUT’ Specifies a maximum line length to be used in the line wrapping process. By default the message to be output is split into chunks of no more than the maximum line length, and each chunk is written on a new line. The split is made at word boundaries if possible. The default maximum line length is 79 characters.

  If VALUE is set to 0, no wrapping will occur. If it is set greater than 6, it specifies the maximum output line length. Note that the minimum VALUE is 7, to allow for exclamation marks and indentation.
- ’STREAM’ Specifies whether or not ERR should treat its output unintelligently as a stream of characters. If VALUE is set to 0 (the default) all non-printing characters are replaced by blanks, and line wrapping occurs (subject to SZOUT). If VALUE is set to 1, no cleaning or line wrapping occurs.
- ’REVEAL’ Allows the user to display all error messages cancelled when ERR_ANNUL is called. This is a diagnostic tool which enables the programmer to see all error reports, even those ’handled’ by the program. If VALUE is set to 0 (the default) annulling occurs in the normal way. If VALUE is set to 1, the message will be displayed.
- ’ENVIRONMENT’ This is not a true tuning parameter name but causes the environment variables associated with all the true tuning parameters to be used if set. If the environment variable is not set, the tuning parameter is not altered. The VALUE argument is not used.

2. The tuning parameters for MSG and ERR operate partially at the EMS level and may conflict in their requirements of EMS.

3. The use of SZOUT and STREAM may be affected by the message delivery system in use. For example there may be a limit on the the size of a line output by a Fortran WRITE and automatic

line wrapping may occur. In particular, a NULL character will terminate a message delivered by the ADAM message system.

4. With REVEAL, messages are displayed at the time of the ANNUL. As REVEAL operates at the EMS level they are displayed with Fortran WRITE statements so, depending upon the delivery mechanism for normal messages, they may appear out of order.

D.3 Deprecated Routine ERR_OUT

Purely for compatibility with previous versions of ERR, the routine ERR_OUT is provided. It should not be used in any new code – usually a call to ERR_REP is all that is required. If it is essential that the message be delivered to the user immediately, ERR_REP should be followed by a call to ERR_FLUSH.

ERR_OUT
Report a message then flush and annul the current error context.

Description:

The message is reported using ERR_REP and then the current error context is flushed and annulled using ERR_FLUSH. On successful completion, STATUS is reset to SAI__OK.

If STATUS is set to SAI__OK on entry, a warning message will be generated but no error results.

If an error occurs during output of the error messages, the error context is not annulled and STATUS is returned set to ERR__OPTER.

Invocation:

```
CALL ERR_OUT( PARAM, TEXT, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
The error message name.

TEXT = CHARACTER * (*) (Given)
The error message text.

STATUS = INTEGER (Given and Returned)
The global status. STATUS should not be SAI__OK on entry; it is reset to SAI__OK on exit unless an output error occurs in which case it will be set to ERR_OPTER.

E Using MSG and ERR within ADAM

E.1 Overview

If the procedures recommended in this document have been adhered to, stand-alone applications which use the Message and Error Reporting Systems can be converted to run within the ADAM environment without change to the message and error reporting code. However, the use of the Message and Error Systems within ADAM applications provides access to facilities which cannot exist in the stand-alone version. This section describes these facilities and how they can be used. In what follows, a prior knowledge of ADAM applications programming to the level of SG/4 (ADAM – The Starlink Software Environment) is assumed.

E.2 Using MSG within ADAM applications

E.2.1 Message parameters

In calls to the MSG output subroutines in ADAM applications, the message name is the name of an ADAM message parameter which can be associated with message text specified in the ADAM interface file instead of the text given in the subroutine argument. Message text specified in an interface file may include message tokens in the normal way.

If the message parameter is not specified in the interface file, the text given in the argument list is used.

Here is an example of using MSG_OUT within an ADAM application:

```
      CALL MSG_OUT( 'RD_TAPE', 'Reading tape.', STATUS )
```

This statement will result in the message:

```
          Reading tape.
```

If the message parameter, ‘RD_TAPE’, is associated with a different text string in the interface module, *e.g.*

```
      message RD_TAPE
         text 'The program is currently reading the tape, please wait.'
      endmessage
```

then the output message would be the one defined in the interface file, *i.e.*

```
      The program is currently reading the tape, please wait.
```

This facility enables ADAM applications to conveniently support foreign languages. However, it is recommended that message parameters used in applications software are not normally defined in the interface file. If message parameters are defined in the interface file of an application, then it is clearly necessary to ensure that the text associated with each message parameter imparts essentially the same information as the text used within the program code, even if they are in different languages.

E.2.2 Parameter references

It is often necessary to refer to an ADAM program parameter in a message. There are two kinds of reference required:

- The keyword associated with a parameter. The keyword is the name for a parameter that the user sees in the documentation of an application. The keyword-parameter association is held in the interface file and is not directly available to the application.
- The name of an object, device or file associated with a parameter. This will usually only be relevant for non-primitive parameters (*e.g.* file names), although primitive parameters (*i.e.* any numerical, CHARACTER or LOGICAL data type) may have an associated object – often their parameter file entry.

Parameter references can be included in the text of a message by prefixing their names with the appropriate escape character. To include the keyword associated with a parameter, its name is prefixed with the percent escape character, "%", *e.g.*

```
      CALL MSG_OUT( 'ET_RANGE', '%ET parameter is ignored.', STATUS )
```

Here, the parameter 'ET' might be associated in the interface file with the keyword "EXPOSURE_TIME", *e.g.*

```
      parameter ET
         type '_INTEGER'
         keyword 'EXPOSURE_TIME'
         prompt 'Exposure time required'
      endparameter
```

In this case, the resultant output would be

```
      EXPOSURE_TIME parameter is ignored.
```

To include the name of an object, device or file associated with a parameter, the parameter name is prefixed with the dollar escape character, "$", *e.g.*

```
      CALL MSG_OUT( 'DS_CREATE', 'Creating $DATASET.', STATUS )
```

If the parameter 'DATASET' is associated with the object called "SWP1234" then this would produce the output

```
      Creating SWP1234.
```

E.2.3 Using program parameters as tokens

It is sometimes necessary to refer to a program parameter in a message, where its name is contained in a variable. In order to incorporate its reference into a message, a message token can be associated with its name. Here is an example of how this would be coded:

```
CALL MSG_SETC( 'PAR', PNAME )
CALL MSG_OUT( 'PAR_UNEXP', '%^PAR=0.0 unexpected', STATUS )
```

In this example, the escape sequence “^” prefixes the token name, 'PAR'. The sequence '^PAR' gets replaced by the contents of the character string contained in the variable PNAME; this, being prefixed by “%”, then gets replaced in the final message by the keyword associated with the parameter.

E.2.4 Reserved token STATUS

The token name “STATUS” is reserved for use by the Error System and should not be used in Message System calls. It will never produce a useful message from an MSG subroutine.

E.2.5 Getting the conditional output level

In addition to the subroutine for setting the filter level for conditional message output, *i.e.* MSG_IFSET, the ADAM version of MSG also provides subroutine MSG_IFGET to get a character string from the parameter system and use this to set the filter level The subroutine has the calling sequence:

```
CALL MSG_IFGET( PNAME, STATUS )
```

where PNAME is the parameter name. It is recommended that one parameter name is used universally for this purpose, namely MSG_FILTER, in order to clarify the interface file entries. The three acceptable strings for MSG_FILTER are:

QUIET – representing MSG__QUIET;

NORMAL – representing MSG__NORM;

VERBOSE – representing MSG__VERB.

DEBUG – representing MSG__DEBUG

Abbreviations are accepted but any other value will result in an error report and STATUS being returned set to MSG__INVIF.

E.2.6 Synchronising message output

Graphical and textual output from an ADAM application can have problems with synchronisation. This is because the message output arrives at the terminal via the command process and is buffered, whereas the graphical output is sent directly to the terminal. The effect of this is that graphical output to the terminal can get corrupted by message output. This problem can be avoided by using the subroutine MSG_SYNC, which flushes the textual output buffer of the command process:

```
      CALL MSG_SYNC( STATUS )
```

MSG_SYNC should be called immediately before any graphical output to avoid any corruption.

E.3 Using ERR within ADAM applications

E.3.1 Error message parameters

In calls to the subroutine ERR_REP in ADAM applications, the error name is the name of an ADAM message parameter which may be associated with message text specified in the interface file instead of the text supplied in the subroutine call. The system operates in the same way as for MSG (see §E.2.1).

E.3.2 Initial error context level

In the non-ADAM, stand-alone, version of the Error System the initial error context level is set to one. Thus, before any calls to ERR_MARK have been made, calls to ERR_LEVEL will return a value of one and any error reports made at this level to be delivered immediately to the user. However, for ADAM applications, error reporting is deferred before the user's application code is called. Thus calls to ERR_LEVEL made before any calls to ERR_MARK will return a value of two to the application,

Any remaining error reports are delivered to the user by the "fixed-part" of the ADAM application after execution of the user's application code.

E.3.3 Reserved token STATUS

Historically, considerable use in ADAM was made of the VAX/VMS MESSAGE utility, both to generate globally unique error codes and to associate each error code with the text of an error message. The utility is also used by the VMS operating system for reporting its own messages. This has resulted in the idea of a reserved message token, STATUS, which would expand to the message associated with the given status value.

Every call to ERR_REP results in the status value given being converted into its associated VAX/VMS message, and then this message is associated with the token STATUS. Hence the global status may be used to construct an error message as follows:

```
      CALL ERR_REP( 'DSCALE_BAD', '%DSCALE: ^STATUS', STATUS )
```

This method of constructing error messages makes the assumption that *all* status values and operating system error numbers form part of a homogeneous set which can easily be associated with some appropriate message text. It is therefore not portable from VAX/VMS to other computer systems. If the STATUS token is used on other systems, an attempt will be made to translate it as a Starlink facility status value.

Subroutines following the recommended error reporting strategy outlined in §3.6 will always make an accompanying error report when the returned status is set to an error value. As a result, error reports based upon the reserved token STATUS are rarely necessary and limit the portability of ADAM software.

New code should not use the reserved token STATUS. If it is required to include the message associated with an error code in an error report, subroutine ERR_FACER should be used to create some other token.

E.3.4 Returning the status to the environment

ADAM applications are Fortran subroutines which are called by the software environment. An ADAM application has the schematic form

```
SUBROUTINE APPLIC( STATUS )

...

<application code>

...

END
```

where the argument STATUS is given as SAI__OK. When the application ends and returns control to the environment, the final value of STATUS will reflect whether or not the application finished on an error condition. This provides the environment with a status to be associated with the application as a whole. The exit status of the application can be used by the environment to decide what to do next. If a status value other than SAI__OK is returned to the ADAM environment, then any pending error messages are output. Applications which fail should therefore return an error status. This recommendation does not, in fact, differ from that for any other subroutine obeying the inherited status strategy see §3.2).

E.4 Using the Error Message Service for ADAM system programming

The Message and Error Systems are intended for use when writing stand-alone and ADAM applications.

For ADAM system programming, where no assumptions may be made about the working order of either the ADAM parameter or data systems, another subroutine library, EMS (the Error Message Service) is available. The subroutine calling sequences for EMS are very similar to those of the Message and Error Systems (indeed many MSG and ERR routines are implemented as straight-through calls to EMS – hence the occurrence of EMS error messages from MSG or ERR routines). EMS is described in SSN/4. The Error Message Service library is intended specifically for use in ADAM system or general low level subroutine libraries. It should not be used in applications software.

E.5 ADAM-special subroutine specifications

MSG_IFGET
Get the filter level for conditional message output from the ADAM parameter system

Description:

Obtain a value for the named parameter and use it to set the value for the filter level for conditional message output. It is recommended that one parameter name is used universally for this purpose, namely MSG_FILTER, in order to clarify the interface file entries. The parameter must be of type _CHAR and the three acceptable strings it may take are:

- QUIET – representing MSG__QUIET;
- NORMAL – representing MSG__NORM;
- VERBOSE – representing MSG__VERB.
- DEBUG – representing MSG__DEBUG.

MSG_IFGET accepts abbreviations of these strings; any other value will result in an error report and the status value being returned set to MSG__INVIF. If an error occurs getting the parameter value, the appropriate status value is returned and an additional error report is made.

Invocation:

```
CALL MSG_IFGET( PNAME, STATUS )
```

Arguments:

PNAME = CHARACTER ∗ (∗) (Given)
The filtering level parameter name.

STATUS = INTEGER (Given and Returned)
The global status.

MSG_SYNC
Synchronise message output via the user interface

Description:

This performs a synchronisation handshake with the user interface. This is required if the current task has been outputting messages via the user interface and now wants to use a graphics cursor on the command device. If a synchronisation error occurs, then an error report is made and the status value is returned set to MSG__SYNER.

Invocation:

```
CALL MSG_SYNC( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status: it is returned set to MSG__SYNER on error.

F Portability

F.1 Overview

This section discusses the portability of MSG and ERR, including the coding standard adopted for MSG and ERR and a list of those Starlink packages which need to be ported to the target machine before a port of MSG or ERR can proceed.

F.2 Coding and porting prerequisites

The standard of Fortran used for the coding of MSG and ERR is fundamentally Fortran 77, using the Starlink Fortran coding conventions described in SGP/16. Several common extensions to the Fortran 77 standard are used in source code for MSG and ERR, they are as follows:

- end-of-line comments using the "!" symbol;
- subprogram names may be longer than six characters (but are always shorter than ten characters);
- subprogram names include the "_" symbol;
- symbolic constant names may be longer than six characters (but are always shorter than eleven characters);
- symbolic constant names may include the "_" symbol;
- the full ASCII character set is assumed in character constants.

To use MSG and ERR on any computer system, the Starlink Error Message Service, EMS (see SSN/4) and Character Handling Routines, CHR (see SUN/40) must be available.

F.3 Operating system specific routines

The following routines have system-specific features and may produce differing results on different platforms. System-specific code (apart from minor format differences on WRITE statements) is all in the equivalent EMS routines.

ERR_FACER Assign a Starlink facility error message to a token.

Calls its EMS counterpart which is system-specific – for more details see SSN/4.

ERR_FIOER Assign a Fortran I/O error message to a token.

Calls its EMS counterpart which will generally need to be rewritten for each new target platform. Current versions have the appropriate messages hardwired into the code. For more details see SSN/4.

ERR_SYSER Assign an operating system error message to a token.

Calls its EMS counterpart which is platform specific. For more details see SSN/4.

ERR_FMTx Assign a value to a message token (formatted).

These routines call their EMS counterpart and behave differently on different systems in the event of an error. Sometimes the errors are reported by Fortran and can be trapped but sometimes they are not. If they are reported, the token remains unset but if they are not reported, the character string generated can differ.

G Calculating Globally Unique Error Codes

Starlink facility error codes will normally be generated by the Starlink MESSGEN utility on UNIX (see SUN/185).

This section presents an alternative method for calculating compatible error status codes for subroutine libraries. In order to be used effectively, it requires a Fortran compiler capable of four byte integer representation. If this is not the case, then the status values generated will *not* be globally unique.

The error codes are calculated using the equation:

$$CODE = 134250498 + 65536\times < fac > + 8\times < mes > \qquad (1)$$

Here, $< mes >$ is the message number (in the range 1 to 4095) assigned to the error condition by the author of the subroutine library, and $< fac >$ is the facility number (in the range 1 to 2047) allocated to this subroutine library. Developers wishing to have facility numbers allocated to subroutine libraries should contact the Starlink Software Librarian (*i.e.* ussc@star.bnsc.rl.ac.uk).

H Changes and New Features in Version 1.4

- New routine ERR_FACER is released to complete the distinction between various types of error code and the messages associated with them. (§3.15 and Appendix D.)
- The Starlink MESSGEN utility is now the preferred way of generating error codes and include files. (§3.3 and Appendix G)
- The deprecated reserved token, STATUS, is now treated as a Starlink facility error message. (§E.3.3)
- This document has been modified to reflect the above changes.

I Changes and New Features in Version 1.4-1

Version 1.4-1 contains no functional changes. It merely updates the makefiles and other scripts according to the latest templates.

This document is revised slightly to remove descriptions of the system on VMS (which is no longer updated). A hypertext version of the document is now available.

J Changes and New Features in Version 1.5

- A preliminary C interface (see Appendix 5) has been added.
- MSG_LOAD has been modified to remove an intermediate buffer which restricted the length of the message which could be handled.
- In `msg_par`, the maximum size of a message (`MSG__SZMSG`) has been increased from 200 to 300 characters.
- In `err_par`, the maximum size of a message parameter name has been corrected to 15 (from 200).
- An installation test for the C interface has been added to the test target in the makefile.

K Changes and New Features in Version 1.6

Subroutine MSG_TUNE and ERR_TUNE have been added to control various aspect of output.

Shared libraries are now produced for Linux as well as Solaris.

This document is revised to add a section on tuning and to improve the look of the hypertext form.

L Changes and New Features in Version 1.7

This version has been substantially revised internally to use version 2.0 of EMS, which is implemented in C.

MERS now uses only the public interface to EMS (formerly a number of internal EMS routines were called).

Because the lowest level is now in C, output of error message at the base error reporting level is now done by C printf commands. On some systems this can lead to incorrect sequencing of messages if other output is done by Fortran WRITE.

A bug which caused the linker to report subroutine MSG_IFSET missing if MSG_IFGET was called, has been corrected.

This section has been added to this document.

M Changes in Version 1.7-1

Corrects a problem using the reserved token STATUS if another token precedes it in the message, and includes a minor update to SUN/104.

N Changes in Version 1.8

MSG1_GREF is completely revised to simplify its interface with the parameter system, using a new routine SUBPAR_GREF. This allows the underlying parameter system to be replaced more easily. Suboutine MSG1_GLOC is now not requited and has been removed. This Version of MERS requires PCS Version 4.1 or later.

O Changes in Version 1.8-1

Builds with the Gnu auto tools and released under the GPL.

P Changes in Version 1.8-2

Added MSG__DEBUG reporting level.