SSN/4.14

Starlink Project
Starlink System Note 4.14

P.C.T. Rees
A.J. Chipperfield
P.W. Draper

31 Jul 2008

Copyright © 2001 Council for the Central Laboratory of the Research Councils
Copyright © 2008 Science and Technology Facilities Council

EMS
Error Message Service
Version 2.2-0
Programmer's Manual

Abstract

This document describes the Error Message Service, EMS, and its use in system software. The purpose of EMS is to provide facilities for constructing and storing error messages for future delivery to the user – usually via the Starlink Error Reporting System, ERR (see SUN/104). EMS can be regarded as a simplified version of ERR without the binding to any software environment (*e.g.* for message output or access to the parameter and data systems). The routines in this library conform to the error reporting conventions described in SUN/104. A knowledge of these conventions, and of the ADAM system (see SG/4), is assumed in what follows.

This document is intended for Starlink systems programmers and can safely be ignored by applications programmers and users.

Copyright © 2001 Council for the Central Laboratory of the Research Councils
Copyright © 2008 Science and Technology Facilities Council

Contents

1 Introduction 1

2 When to Use the Error Message Service 1

3 Reporting Errors 2

4 Message Tokens 2

5 Reporting Status, Fortran I/O and Operating System Errors 3

6 Message Output 4
- 6.1 No EMS_FLUSH 4
- 6.2 EMS without an environment 4
- 6.3 EMS within an environment 4

7 Intercepting Messages 5

8 C Language Interface 6

9 Compiling and Linking with the Error Message Service 7

10 References 8

A Justification for EMS 9

B Using EMS together with POSIX threads 10

C Using EMS within ADAM system software 12
- C.1 Overview 12
- C.2 Message parameters 12
- C.3 Parameter references 12
- C.4 Reserved tokens 12
- C.5 Synchronising message output 12
- C.6 Routines specific to the ADAM fixed part 13
  - ERR_CLEAR 14
  - ERR_START 15
  - ERR_STOP 16

D Portability 18
- D.1 Porting prerequisites 18
- D.2 Operating system specific routines 18

E Include Files 19

F Fortran Subroutine List 20

G C Interface Function Prototypes 22

H Fortran Subroutine Specifications 24

EMS_ANNUL 25
EMS_BEGIN 26
EMS_ELOAD 27
EMS_END 28
EMS_EXPND 29
EMS_FACER 30
EMS_FIOER 31
EMS_GTUNE 32
EMS_LEVEL 33
EMS_MARK 34
EMS_MLOAD 35
EMS_RENEW 36
EMS_REP 37
EMS_RLSE 38
EMS_SETx 39
EMS_STAT 40
EMS_SYSER 41
EMS_TUNE 42

List of Figures

1 The relationship between MSG, ERR and EMS 1
2 Overview of an application 9

1 Introduction

The Error Message Service is an implementation of the error reporting conventions discussed in SUN/104 which is not dependent on any particular software environment and can therefore be used in 'stand-alone' programs and for implementing components of software environments. In order to allow its use in environment system software, EMS provides no callable facility for delivering these messages to the user. It just provides facilities for constructing and storing reported error messages. In doing so, it also provides many of the facilities required by MSG and ERR. Indeed, in many cases the MSG and ERR routines are implemented as straight-through calls to EMS routines.

EMS can also be used in the threaded parts of applications and libraries when compiled with POSIX threads support, see Appendix B.

The hierarchical relationship between MSG, ERR and EMS is illustrated in *Figure 1*.

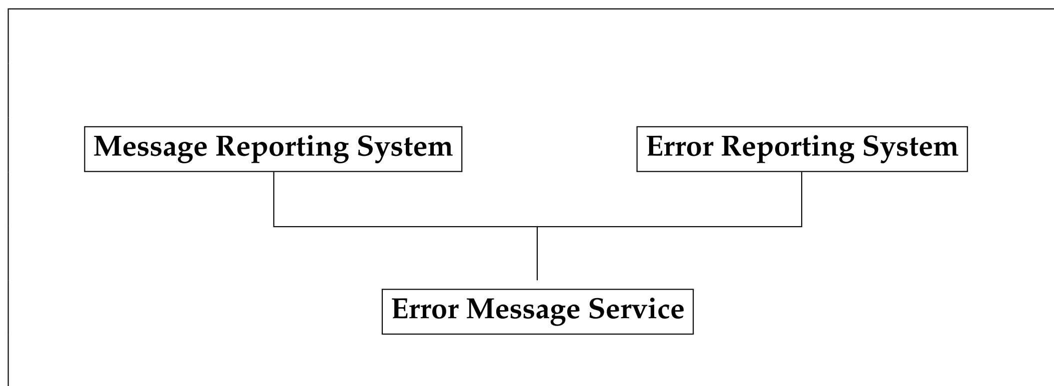

Figure 1: The relationship between MSG, ERR and EMS

EMS was originally written in Fortran and a C interface provided but since Version 2.0 it is written in C and a Fortran interface provided. This document concentrates on the Fortran interface. The C interface is described briefly – the way it is used can be inferred from the use of the Fortran interface.

2 When to Use the Error Message Service

At the level of applications code the Error Reporting System, ERR, exclusively should be used for error reporting. Within environment subroutine libraries the rule is to use calls to EMS unless it is necessary to deliver error messages to the user (*i.e.* by calls to `ERR_FLUSH`). A subroutine library *must* use one or other of these error reporting libraries exclusively: *i.e.* if EMS calls are used within a subroutine library, they must be used exclusively throughout the library – no calls to ERR routines are allowed. Furthermore, if ERR calls are used to report errors within a subroutine library, say because calls to `ERR_FLUSH` are required during interaction with the user, then it must be understood that this subroutine library exists at a hierarchical level above the environment system software. This subroutine library cannot then be called from other libraries

purporting to exist at lower levels within the subroutine hierarchy – to do so could result in unpredictable behaviour when an error occurs.

An exception to the above is when working with a threaded application. For these EMS should be used in the threaded portions and in general any error reporting via ERR should be deferred until the threaded sections complete. This avoids problems with global variable access in the ERR and ADAM libraries. See Appendix B.

3 Reporting Errors

The subroutine used to report errors is `EMS_REP`. It has a calling sequence analogous to that of `ERR_REP`, *i.e.*

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

The argument `PARAM` is the error message name, `TEXT` is the error message text and `STATUS` is the inherited status. The `TEXT` and `STATUS` arguments are the same as those used in calls to `ERR_REP` and are discussed fully in SUN/104.

On exit from `EMS_REP` the status argument remains unchanged, with two exceptions:

- If `STATUS` set to SAI__OK on entry, `STATUS` is returned set to EMS__BADOK.
- If an output error occurs, `STATUS` is returned set to EMS__OPTER.

Appendix E lists the symbolic constants for all the possible EMS errors.

The deferral of error reporting can be controlled within EMS: `EMS_MARK` and `EMS_RLSE` respectively mark and release error reporting contexts; `EMS_ANNUL` annuls all error reports pending output in a given error context; `EMS_BEGIN` and `EMS_END` respectively begin and end error reporting environments. The behaviour of these subroutines is analogous to `ERR_MARK`, `ERR_RLSE`, `ERR_ANNUL`, `ERR_BEGIN` and `ERR_END` provided by the Error Reporting System (see SUN/104). EMS does not provide an analogue to `ERR_FLUSH`.

4 Message Tokens

The facility to use message tokens to embed the values of program variables within the message text is available in EMS. As in MSG and ERR, these message tokens are indicated by prefixing the token name with an up-arrow, “^”, escape character within message text, *e.g.*

```
CALL EMS_REP( 'EMS_ROUTN_TOKEN', 'Error text: ^TOKEN', STATUS )
```

Message tokens can be set to the most concise character encoding of a given variable’s value with the routines `EMS_SET`*x*, *e.g.*

```
CALL EMS_SETC( 'TOKEN', 'A token' )
```

For the EMS_SETx routines, x corresponds to one of five standard Fortran 77 data types:

x	*Fortran Type*
D	DOUBLE PRECISION
R	REAL
I	INTEGER
L	LOGICAL
C	CHARACTER

These routines are analogous to the subroutines MSG_SETx provided by the Message Reporting System (see SUN/104). As for the MSG routines, if the token has already been set, the new string will be appended to the existing one. All defined message tokens are annulled by a call to EMS_ANNUL, EMS_ELOAD, EMS_EXPND, EMS_REP or EMS_MLOAD[1]. They may be renewed to their previous defined values by a call to EMS_RENEW provided that no new tokens have been defined in the meantime.

5 Reporting Status, Fortran I/O and Operating System Errors

Three subroutines are provided by the Error Message Service to enable a message token to be built from the error flag value returned from standard Starlink library routines, system routines or Fortran I/O operations. (*For C programmers, emsErrno is also provided, see Section 8.*) It is important that the correct routine is called, otherwise the wrong message or, at best, only an error number will be obtained.

It should be noted that the messages returned using these subroutines will depend upon the operating system in use. The error messages returned by operating systems vary considerably in their clarity and so should not be relied upon as the sole source of information in error reports.

The subroutines are:

```
EMS_FACER( TOKEN, STATUS )
```

where STATUS is a standard Starlink facility status value,

```
EMS_FIOER( TOKEN, IOSTAT )
```

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

```
EMS_SYSER( TOKEN, SYSTAT )
```

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

In each case the message associated with the given error code is assigned to the specified message token. On completion, each of these subroutines returns its second argument unchanged.

[1] EMS_MLOAD is now deprecated, EMS_EXPND should be used instead

6 Message Output

6.1 No EMS_FLUSH

At the risk of belabouring the point, the Error Message Service provides no callable facility for delivering messages to the user, *i.e.* there is no analogue to `ERR_FLUSH` provided by EMS for the programmer. This is a direct result of the binding of ERR to the particular software environment being used. In the Error Message Service no such binding can be used reliably. EMS is only able to stack and annul error messages and to control error deferral using the same error table as the ERR subroutine library. Responsibility for the delivery of error messages to the user therefore rests at a higher level, ultimately with the application programmer.

Message delivery will usually be achieved by calling `ERR_FLUSH` at a higher level, or by 'Message Interception' (see Section 7) but EMS has another feature which can be helpful in 'stand-alone' programs, *i.e.* without an environment present.

6.2 EMS without an environment

Outside any environment EMS has to have some reliable mechanism for ultimately delivering any reported error messages to the user, since it cannot be guaranteed that the ERR library is being used at the level of the application. This is achieved by the having the initial error context level of EMS deliver any reported error message immediately to the user using a C `printf` statement: *i.e.* if all an application does is call `EMS_REP`, then all error messages are immediately delivered to the user when they are reported. Any higher error context level, set with a call to `EMS_MARK` or `EMS_BEGIN`, will have the effect of deferring message output until the error reporting context is returned to its initial level with calls to `EMS_RLSE` or `EMS_END`. Any deferred error messages will then be delivered to the user and any further error messages reported in this context will again be delivered immediately to the user.

The behaviour of EMS without an environment present means that it is essential for error reporting contexts to be properly nested within a package and for a subroutine library using EMS not to exit at a context level other than the one at which it was invoked. If context levels are incorrectly nested within a package it can lead to EMS failing to deliver reported error messages when an application goes wrong. Because the nested use of `EMS_MARK` and `EMS_RLSE` (or `EMS_BEGIN` and `EMS_END`) is normally confined within the same routine, incorrect nesting of context levels will normally result from simple programming errors which can be easily traced. However, there is no simple safeguard against the incorrect nesting of error context levels provided by EMS: thorough testing is the most effective approach.

> *N.B. On some systems, C and Fortran output cannot be reliably interspersed. This fallback output at the EMS initial context level should therefore not be used in programs which also use Fortran* `WRITE` *statements.*

6.3 EMS within an environment

When using EMS from within an environment, the environment should ensure that the delivery of any reported error messages within the environment is deferred and that any undelivered

messages are output to the user when the application has finished. There are three routines (actually part of the ERR library), exclusively for the use of environment implementors, which facilitate this task – they are discussed in Appendix C.

7 Intercepting Messages

Although there is no callable provision for message output directly from EMS, the output of messages at this level in the ADAM subroutine hierarchy is still possible. It can be done by intercepting any pending error messages within the current error context (or informational messages) using subroutines `EMS_ELOAD` or `EMS_EXPND`, which return the message in a character variable, and then using a private package-related mechanism to output the message. Here, the mechanism used to output the message, *e.g.* to a log file, and its resilience to failure must be the responsibility of the subroutine library calling EMS. The use of private package equivalents to `MSG_OUT` or `ERR_FLUSH` must be fully justified and the justification presented in the documentation for the subroutine library concerned.

`EMS_ELOAD` returns error message pending in the current error context, one by one in a series of calls; it has the calling sequence:

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

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. If there are no messages pending on the first call, a warning message is returned with `STATUS` set to EMS__NOMSG.

After `STATUS` has been returned SAI__OK, the whole process is repeated for subsequent calls. This process is the same as for `ERR_LOAD` (see SUN/104) which just calls `EMS_ELOAD`.

`EMS_EXPND` has the calling sequence:

```
CALL EMS_EXPND( TEXT, OPSTR, OPLEN, STATUS )
```

`TEXT` is the message text, `OPSTR` is the returned message text, `OPLEN` is the length of the message in `OPSTR` and `STATUS` is the inherited status.

The behaviour of `EMS_EXPND` is to expand any message tokens in the message text and return the expanded message through the character variable `OPSTR`.

For both `EMS_EXPND` and `EMS_ELOAD`, 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 symbolic constants EMS__SZPAR and EMS__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 `EMS_PAR` (see Appendix E).

8 C Language Interface

To enable EMS to be used in C-only programs, without the need to link with Fortran libraries, it has now been implemented entirely in C. A C function equivalent exists for each of the Fortran EMS routines described in this document except `EMS_FIOER` which is specific to the Fortran language and has therefore been omitted.

The current naming scheme for the C functions (`emsAnnul` *etc.*) replaces the earlier one (`ems_annul_c` *etc.*), but the old names will still be recognised. There is no change to the argument lists. (See Appendix G for more details.)

Additional functions are available for C programmers:

- `emsErrno( char const *token, int errval )` This will assign the message associated with a C errno (`errval`) to the specified token. Note that this is logically different from `emsSyser`; on UNIX they produce the same result but on other systems they may be different.
- `int emsGtune( char const *key, int *status )` This returns the value of a tuning parameter. If differs from the Fortran version as the value is returned, not passed back by reference.
- `emsSetnc( char const *token, char const *string, int maxchar )` This is similar to `emsSetc` but will limit the token length to `maxchar` characters. It must be used if `string` is not null-terminated. `ems.h` defines `ems_setc_c` to be equivalent to `emsSetnc` - there is no Fortran interface for this function.
- `int emsStune( char const *key, int value, int *status )` This replaces the `emsTune` function. It returns the old value of the given parameter so that it can be re-established using another call to this function (this may be necessary for systems that do not share tuning values).

A full list of the C function prototypes is provided in Appendix G.

The function `emsExpnd` has the argument `maxlen`, not found in its equivalent Fortran call sequence. This argument represents the maximum allowable string length for the expanded message and is necessary for the use of returned C character strings where the declared length cannot be determined. Normally, the argument `maxlen` is given the global constant value EMS__SZMSG. There should be space for `maxlen`+1 characters in the output string.

The correspondence between ANSI Fortran 77 data types and ANSI C data types is not defined: *i.e. it is implementation dependent*. However, the most likely correspondence can be assumed and this has been coded into the C/Fortran interface. This correspondence is apparent from the C interface function prologues provided in Appendix G and is summarised in the following table:

C Type	*Fortran Type*
double	DOUBLE PRECISION
float	REAL
int	INTEGER
int	LOGICAL
char	CHARACTER

Note that the interpretation of the `int lvalue` argument presented to the function `emsSetl` is that defined by the ANSI C language.

9 Compiling and Linking with the Error Message Service

There are three Fortran include files available for use with the Error Message Service: `SAE_PAR`, `EMS_PAR` and `EMS_ERR` (see Appendix E 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 'EMS_PAR'
```

Equivalent header files are installed in `/star/include` for use in C code which is calling EMS – they are named `sae_par.h`, `ems_par.h` and `ems_err.h`. In addition, `ems.h` contains the function prototypes for each of the C functions. The syntax

```
#include "sae_par.h"
#include "ems.h"
```

should be used within the C code, and the compiler option `-I/star/include` used when compiling C and Fortran code.

The Error Message Service is included automatically when programs are linked using the ADAM application linking commands, `alink` *etc*.

To link a non-ADAM Fortran program with the the Error Message Service, the command line would be something like:

```
% f77 -I$STARLINK_DIR/include program.f -L$STARLINK_DIR/lib `ems_link` \
      -o program.out
```

The command to compile and link a C program might be:

```
% cc -o program -I$STARLINK_DIR/include -L$STARLINK_DIR/lib program.c \
     `ems_link`
```

> *Note that the commands used to invoke the C and Fortran compilers vary from one UNIX implementation to another (indeed, there may be more than compiler of each type on the same machine).*

If the C program makes no use of the EMS Fortran interface, the `‘ems_link‘` command may be changed to `‘ems_link Conly‘`. This will avoid the linker looking for the Starlink CNF library, which may not be present at a non-Starlink site.

10 References

Note: Only the first author is listed here.

Lawden, M.D.	SG/4 — ADAM – The Starlink Software Environment.
Rees, P.C.T.	SUN/104 — MSG and ERR – Message and Error Reporting Systems.
Chipperfield, A.J.	SUN/185 — MESSGEN – Starlink Facility Error Message Generation.

A Justification for EMS

The hierarchical structure of an application within a software environment is illustrated in *Figure 2*.

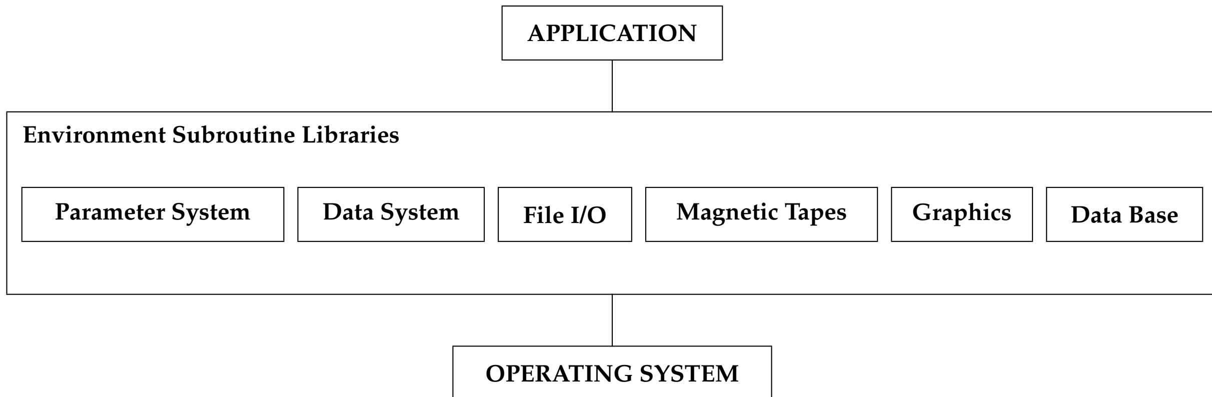

Figure 2: Overview of an application

It can be seen from *Figure 2* that between an application and the operating system any environment is realised as a number of subroutine libraries, providing a set of documented facilities for the applications programmer to use. Between the user and an application, the environment will normally also provide a user interface which controls the loading of specific applications, the I/O to the user and the parameter system.

The exclusion of any user-callable message delivery mechanism from the Error Message Service allows EMS to be used for error reporting within software environment system routines without the danger of recursion on error. Conversely, the Error and Message Reporting Systems, ERR and MSG (see SUN/104), have bindings to specific software environments and thus the ability to deliver error and informational messages to the user via the user interface of that environment. The binding of ERR and MSG to a given software environment precludes their use for error reporting from within the environment system software because the error reporting would rely to some (probably unpredictable) extent upon the working state of that environment. Thus, ERR and MSG act as interfaces for a given environment with EMS, to be used only within applications software.

As an example, the ADAM versions of ERR and MSG are able to obtain the message text associated with a given message parameter from the ADAM interface file and are also able to include ADAM parameter keywords and references (*i.e.* file names, HDS object names or device names) in the message text. These features are useful when reporting error and informational messages from ADAM applications software. However, in order to provide these facilities, MSG and ERR make calls to the ADAM parameter system, data system and user interface and so depend upon the working state of these parts of the ADAM environment. These features preclude the use of MSG and ERR in the development of any subroutine library used by the parameter system, data system or user interface.

B Using EMS together with POSIX threads

When built using the `-with-pthreads` configuration option, which is the default on platforms supporting POSIX threads, EMS can be used in a threaded application or library. The only requirement is that the sections of the application that use threads are wrapped in `emsMark/emsRlse` blocks, as in:

```
void *make_report( void *threadid )
{
    int tid;
    int status = SAI__ERROR;

    tid = (int) threadid;
    emsSeti( "THREADID", tid );
    emsRep( "THREAD", "Oh no an error in thread ^THREADID", &status );
    return NULL;
}

int main ( int argc, char *argv[] )
{
    pthread_t threads[ NUM_THREADS ];
    int rc;
    int t;
    int status = SAI__OK;

    emsMark();

    for ( t = 0; t < NUM_THREADS; t++ ) {
        pthread_create( &threads[ t ], NULL, make_report, (void *)t );
    }

    /*  Wait for threads to complete. */
    for ( t = 0; t < NUM_THREADS; t++ ) {
        pthread_join( threads[ t ], NULL );
    }

    /* Check for the exit status */
    emsStat( &status );

    emsRlse();
}
```

The error messages can then be delivered or loaded as normal (when using ERR it is important not to make any calls to it in the threaded section, as this may access unprotected global data). The status return recovered by `emsStat` will be one from one of the threads, it is not possible to find out the status from all threads using EMS. In a standalone program as shown the error messages:

```
!! Oh no an error in thread 1
!  Oh no an error in thread 2
!  Oh no an error in thread 3
!  Oh no an error in thread 4
```

will be shown using the default delivery `printf` mechanism during the `emsRlse` call (since the error stack context falls to the base level).

C Using EMS within ADAM system software

C.1 Overview

This document has so far discussed the use of EMS in a general context. In this section, specific differences between EMS and the ADAM versions of the Message and Error Reporting Systems, MSG and ERR, are presented.

C.2 Message parameters

In calls to the ADAM version of `ERR_REP`, the error message name `PARAM` is a globally unique identifier for the error message text which corresponds to the message parameter of the error message. The ADAM interface file can be used to associate this message parameter with an error message, and only if this message parameter is not defined in the interface file is the argument `TEXT` used in the error report. When using `EMS_REP` (or the deprecated `EMS_MLOAD`) within the ADAM environment software, no access to the ADAM interface file exists and so the error message given by the argument `TEXT` is always used. The use of a globally unique error message name is still recommended for reported error messages to uniquely identify the error message text to the programmer, using the convention outlined in SUN/104.

C.3 Parameter references

The ADAM versions of MSG and ERR allow reference to ADAM program parameters within the message text. Reference can be made either to parameter-keyword associations or to the name of an ADAM data system object, data file or device associated with a parameter. In message text used in calls to MSG and ERR routines these parameter associations are indicated by the percent, "%", and dollar, "$", escape characters respectively. EMS does not have access to the ADAM parameter system and so cannot refer to ADAM parameters in the text of messages. The escape characters "%" and "$" are therefore treated literally as part of the message text in calls to `EMS_REP` and `EMS_EXPND` (or the deprecated `EMS_MLOAD`), as they are in the stand-alone versions of MSG and ERR.

C.4 Reserved tokens

In the ADAM versions of MSG and ERR a reserved token, 'STATUS', can be used to insert the message associated with a status value. In EMS, as in the stand-alone versions of MSG and ERR, there are no reserved message tokens

C.5 Synchronising message output

Because there is no callable mechanism provided by EMS to deliver messages to the user, no message synchronising mechanism is provided by EMS.

C.6 Routines specific to the ADAM fixed part

There are three ERR routines which start up, clear and stop the Error Reporting System within the ADAM fixed part to ensure that the delivery of any reported error messages is deferred and that any messages pending output are delivered to the user when the application has finished. These subroutines are `ERR_START`, `ERR_CLEAR` and `ERR_STOP` – they are discussed in this document because they are purely for system programming and will affect the use of EMS as well as ERR within ADAM.

`ERR_START` has the effect of marking a new EMS context and thus deferring the delivery of any reported error messages to the user. The new context is known as the default context.

`ERR_CLEAR` has the effect of returning the current error context to the 'default' level set by `ERR_START` and flushing any reported error messages to the user. Thus, the effect of any mismatched 'mark' and 'release' calls within an application is annulled.

`ERR_STOP` clears the error message table by calling `ERR_CLEAR` and then returns the error reporting context to the initial context level, *i.e.* the level prior to the call to `ERR_START`.

> *The use of these three routines is entirely reserved for starting up, clearing and stopping error reporting within the environment, e.g. the ADAM fixed part, and must not be used for any other purpose.*

These routines exist only in the ADAM version of the ERR library and are linked using the procedure given in SUN/104.

ERR_CLEAR
Return the error table to the default context and flush its contents

Description:

The Error Reporting System is returned to its default context level and any pending messages are flushed. This routine effectively resets the Error Reporting System:

- unlike `ERR_FLUSH`, no 'faulty application' error message is reported if it is called when there are no error messages pending output, or if it is called with the status value set to SAI__OK;
- the error table is always annulled by a call to `ERR_CLEAR`, irrespective of any message output errors which may occur.

On exit, the status is always returned as SAI__OK.

Invocation:

```
CALL ERR_CLEAR( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The global status.

Implementation Notes :

This subroutine is for use only with the ADAM implementation of the Error Reporting System.

ERR_START
Initialise the Error Reporting System

Description:

Initialise the Error Reporting System and set a new context level (the default level) to defer error message delivery.

Invocation:

```
CALL ERR_START
```

Implementation Notes :

This subroutine is for use only with the ADAM implementation of the Error Reporting System.

ERR_STOP
Close the Error Reporting System

Description:

Flush any messages pending output and return the Error Reporting System to its initial state.

Invocation:

```
CALL ERR_STOP( STATUS )
```

Arguments:

STATUS = INTEGER (Given)

The global status.

Implementation Notes :

This subroutine is for use only with the ADAM implementation of the Error Reporting System.

D Portability

D.1 Porting prerequisites

The Fortran interface to EMS makes use of the Starlink mixed language programming package, CNF (see SUN/209), which must be available for building EMS and programs which use its Fortran interface.

The correct operation of `EMS_FACER` (and `emsFacer`) also requires the necessary facility message files to be installed (see 'Operating system specific routines' below).

D.2 Operating system specific routines

EMS is coded in ANSI C, according to the Starlink C coding conventions described in SGP/4. However, the following routines have system-specific features and may need re-implementing for new platforms. They may also produce differing results on different platforms.

`EMS_FACER` Assign the message associated with a Starlink status value to a *token.*

On UNIX this is implemented using the MESSGEN system (see SUN/185) and thus relies on the required Starlink facility message files having been entered into the system.

When `EMS_FACER` is called to to obtain the message associated with a given `STATUS` value, it first works out the facility number from the `STATUS` value. It then searches for a file with the name `fac_`*facnum*`_err` (where *facnum* is the facility number). If environment variable `EMS_PATH` is defined, it is taken to be a directory search path for the file; if not, a directory search path of `../help` relative to each of the directories on the user's `PATH` is used.

If the file cannot be found along the selected search path, or a message cannot be associated for any other reason, a 'FACERR' message is substituted giving as much information as possible. For example:

ident`, error 147358667 (fac=200,messid=121)`

where the given status value is 147358667 and 200 and 121 are the facility number and message number derived from it.

ident may be one of the following:

`FACERR__BADARG` – the given status value was not a valid Starlink status value.

`FACERR__NOFAC` – the required facility message file was not found.

`FACERR__BADFIL` – the facility message file was faulty.

`FACERR__NOMSG` – the required message number was not found in the facility message file.

`EMS_FIOER` Assign a Fortran I/O error message to a token.

This will generally need to be rewritten for each new target platform. Current versions have the appropriate messages hardwired into the code.

`EMS_SYSER` Assign an operating system error message to a token.

This may need to be rewritten for a new target platform. At the moment UNIX platforms use the same code which accesses `sys_errlist`.

`emsErrno` Assign the message associated with a C errno to a token.

On UNIX platforms this accesses `sys_errlist` to obtain the message.

E Include Files

The symbolic constants defined by the three include files used with EMS are listed below. The way in which these files are included within Fortran or C code is described in Section 9.

`SAE_PAR` (`sae_par.h`) Defines the non-specific status codes for Starlink.

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

`EMS_PAR` (`ems_par.h`) Defines the Error Message Service constants.

- `EMS__BASE` – The base context number.
- `EMS__MXMSG` – Maximum number of messages.
- `EMS__MXOUT` – Maximum length of output text.
- `EMS__SZMSG` – Maximum length of error message text.
- `EMS__SZPAR` – Maximum length of error message name.
- `EMS__SZTOK` – Maximum length of message token text.
- `EMS__TOKEC` – Message token escape character.

`EMS_ERR` (`ems_err.h`) Defines the Error Message Service error codes.

- `EMS__BADOK` – Status set to SAI__OK in call to `EMS_REP` (improper use of `EMS_REP`).
- `EMS__BDKEY` – Invalid keyword will be ignored (improper use of `EMS_TUNE` or EMS_GTUNE).
- `EMS__BTUNE` – Conflicting actions (improper use of `EMS_TUNE`).
- `EMS__CXOVF` – Error context stack overflow (EMS fault).
- `EMS__EROVF` – Error message stack overflow (EMS fault).
- `EMS__NOENV` – Error encountered reading environment variable `EMS_TUNE`.
- `EMS__NOMSG` – No error messages pending output.
- `EMS__NSTER` – Error in nested calls to `EMS_BEGIN` and `EMS_END` (improper use of EMS).
- `EMS__OPTER` – Error encountered during message output.
- `EMS__UNSET` – No status associated with message (improper use of `EMS_REP`).

Notes:

(1) EMS__BADOK is the status returned if `EMS_REP` is called with status SAI__OK whereas EMS__UNSET is the status associated with any messages reported in this way and may therefore be returned by, for example, `EMS_STAT`.

(2) EMS__NOENV is provided for future developments – it is not used at the moment.

F Fortran Subroutine List

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

EMS_BEGIN (STATUS)
Begin a new error reporting environment.

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

EMS_END (STATUS)
End the current error reporting environment.

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

EMS_FACER (TOKEN, STATUS)
Assign the message associated with a Starlink status to a token.

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

EMS_GTUNE (KEY, VALUE, STATUS)
Inquire an EMS tuning parameter.

EMS_LEVEL (LEVEL)
Inquire the current error context level.

EMS_MARK
Start a new error context.

EMS_MLOAD (PARAM, TEXT, OPSTR, OPLEN, STATUS)
(Deprecated) *Expand and return a message.*

EMS_RENEW
Renew any annulled message tokens in the current context.

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

EMS_RLSE
Release (i.e. end) the current error context.

EMS_SETx (TOKEN, VALUE)
Concise encoding of a given value.

EMS_STAT (STATUS)
Inquire the last reported error status.

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

EMS_TUNE (KEY, VALUE, STATUS)
Set an EMS tuning parameter.

G C Interface Function Prototypes

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

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

void emsEload (char ∗parstr, int ∗parlen, char ∗opstr, int ∗oplen, int ∗status);
Return messages from the current error context.

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

void emsErrno (const char ∗token, int errval);
Assign the message associated with a C errno to a token.

void emsExpnd
(const char ∗text, char ∗opstr, const int maxlen, int ∗oplen, int ∗status);
Expand and return a message.

void emsFacer (const char ∗token, int facerr);
Assign the messages associated with a Starlink status value to a token.

int emsGtune (const char ∗key, int ∗status);
Get the value of a tuning parameter

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

void emsMark (void);
Start a new error context.

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

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

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

void emsRlse (void);
Release (i.e. end) the current error context.

void emsSetc (const char ∗token, const char ∗cvalue, ...);
Concise encoding of a given null-terminated string. String `cvalue` must be null-terminated. (For historical reasons, additional arguments are permitted but have no effect.)

void emsSetnc (const char ∗token, const char ∗cvalue, int maxchar);
Concise encoding of a given CHARACTER value - maximum `maxchar` characters.

void emsSetd (const char *token, double dvalue);
Concise encoding of a given DOUBLE PRECISION value.

void emsSeti (const char *token, int ivalue);
Concise encoding of a given INTEGER value.

void emsSetl (const char *token, int lvalue);
Concise encoding of a given LOGICAL value.

void emsSetr (const char *token, float fvalue);
Concise encoding of a given REAL value.

void emsStat (int *status);
Inquire the last reported error status.

int emsStune (const char *key, int value, int *status);
Set an EMS tuning parameter, returning previous value.

void emsSyser (const char *token, int systat);
Assign an operating system error message to a token.

void emsTune (const char *key, int value, int *status);
(Deprecated use emsStune) Set an EMS tuning parameter.

H Fortran Subroutine Specifications

EMS_ANNUL
Annul the contents of the current error context

Description:

Any pending error messages for the current error context are annulled, i.e. deleted. The values of any existing message tokens become undefined and the value of the status argument is reset to SAI__OK.

Invocation:

```
CALL EMS_ANNUL( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The global status: set to SAI__OK on return.

EMS_BEGIN
Begin 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 `EMS_BEGIN` is called with the status argument 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, then an error report to this effect will be made on behalf of the calling application.

Invocation:

```
CALL EMS_BEGIN( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status: set to SAI__OK on return.

EMS_ELOAD
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 in `OPSTR`. 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:

```
CALL EMS_ELOAD( 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.

EMS_END
End the current error reporting environment

Description:

Check if any error messages are pending output in the previous error reporting context. If so, then annul and release the current context; if not, then just release the current context. Return the last reported status value.

Invocation:

```
CALL EMS_END( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The global status.

EMS_EXPND
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 status argument is not set to SAI__OK on entry, no action is taken except that the values of any existing message tokens are always left undefined after a call to `EMS_EXPND`. If the expanded message is longer than the length of the supplied character variable, the message is terminated with an ellipsis.

Invocation:

```
CALL EMS_EXPND( MSG, OPSTR, OPLEN, STATUS )
```

Arguments:

MSG = 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)
The global status.

EMS_FACER
Assign the message associated with a Starlink status value to a token

Description:

The text of the error message associated with the Starlink status value, `STATUS`, is assigned to the named message token. This token may then be included in an error message. If the token is already defined, the error message is appended to the existing token value.

Invocation:

```
CALL EMS_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 vary slightly between systems. For more information, see Appendix D.2.

EMS_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. This token may then be included in an error message. If the token is already defined, the error message is appended to the existing token value.

Invocation:

```
CALL EMS_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 upon the computer system in use.

EMS_GTUNE
Inquire an EMS tuning parameter

Description:

Return the value of an EMS tuning parameter. The possible tuning parameters are those defined in `EMS_TUNE`.

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 `STATUS` is SAI__OK on entry and the routine fails to complete, `STATUS` will be set and an error report made.

Invocation:

```
CALL EMS_TUNE( KEY, VALUE, STATUS )
```

Arguments:

KEY = CHARACTER * (*) (Given)
The tuning parameter keyword

VALUE = INTEGER (Returned)
The current value of the tuning parameter.

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

EMS_LEVEL
Inquire the current error context level

Description:

Return the number of context markers set in the error message table. Any returned value greater than one indicates that the delivery of reported error messages is deferred.

Invocation:

```
CALL EMS_LEVEL( LEVEL )
```

Arguments:

LEVEL = INTEGER (Returned)

The error context level.

EMS_MARK
Start a new error context

Description:

Begin a new error reporting context so that delivery of subsequently reported error messages is deferred and the messages held in the error table. Calls to `EMS_ANNUL` or `EMS_ELOAD` will only annul or return the contents of the error table within this new context.

Invocation:

```
CALL EMS_MARK
```

EMS_MLOAD
(Deprecated) 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 status argument is not set to SAI__OK on entry, no action is taken except that the values of any existing message tokens are always left undefined after a call to `EMS_MLOAD`. If the expanded message is longer than the declared length of the supplied character variable, the message is terminated with an ellipsis.

Invocation:

```
CALL EMS_MLOAD( PARAM, MSG, OPSTR, OPLEN, STATUS )
```

Arguments:

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

MSG = 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)
The global status.

Notes:

This routine is now deprecated as the `PARAM` argument is redundant and the C interface gives no limit for the expanded string size. `EMS_EXPND` should be used instead.

EMS_RENEW
Renew any annulled message tokens in the current context

Description:

Any message tokens which have been annulled by a call to `EMS_REP`, `EMS_EXPND`, `EMS_MLOAD`, `EMS_ANNUL` or `EMS_ELOAD` in the current context are renewed. If any new token value has been defined since the previous tokens were annulled (*e.g.* using the `EMS_SETx` routines), no action is taken. The intended use of `EMS_RENEW` is to renew all existing message tokens immediately after a message has been issued so that they can be re-used in a subsequent message.

Invocation:

```
CALL EMS_RENEW
```

EMS_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. The values associated with any existing message tokens are left undefined. On successful completion, the global status is returned unchanged; if the status argument is set to SAI__OK on entry, an error report to this effect is made on behalf of the application and the status argument is returned set to EMS__BADOK; if an output error occurs, the status argument is returned set to EMS__OPTER.

Invocation:

```
CALL EMS_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: this is left unchanged on successful completion, and is returned set to an EMS__ error if an internal error occurs.

EMS_RLSE
Release (end) an error context

Description:

Release a "mark" in the error message table, returning the Error Message Service to the previous error context. Note that 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 EMS_RLSE
```

EMS_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:

x	*Fortran Type*
D	DOUBLE PRECISION
R	REAL
I	INTEGER
L	LOGICAL
C	CHARACTER

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

Invocation:

```
CALL EMS_SETx( TOKEN, VALUE )
```

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.

EMS_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 EMS_STAT( STATUS )
```

Arguments:

STATUS = INTEGER (Returned)

The last reported error status within the current error context.

EMS_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. This token may then be included in an error message. If the token is already defined, the error message is appended to the existing token value.

Invocation:

```
CALL EMS_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 upon the computer system in use.

EMS_TUNE
Set an EMS tuning parameter

Description:

The value of the EMS tuning parameter is set appropriately, according to the value given. `EMS_TUNE` may be called multiple times for the same parameter. The following keywords and values are permitted:

'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.

'MSGDEF' Specifies the default error reporting level. That is a level below which `EMS_RLSE` will not go. It can therefore be used by environments such as ADAM to prevent any output by EMS due to unmatched marks and releases. This keyword should not be used in other cases.

'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 `EMS_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.

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 `STATUS` is SAI__OK on entry and the routine fails to complete, `STATUS` will be set and an error report made.

Invocation:

```
CALL EMS_TUNE( KEY, VALUE, STATUS )
```

Arguments:

KEY = CHARACTER * (*) (Given)
The tuning parameter keyword

VALUE = INTEGER (Given)
The value to be assigned (see Description)

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

Notes:

(1) SZOUT applies to the case where EMS does its own output - *i.e.* when messages are reported or impending when EMS is at its base level. Normally output will be from the ERR package and is tuned by `ERR_TUNE`.

(2) Some aspects of output for both ERR and MSG are controlled by EMS and its tuning parameters therefore `ERR_TUNE` and `MSG_TUNE` call this subroutine to set the EMS tuning parameters appropriately. This may result in interference.

(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.