ANGLO-AUSTRALIAN OBSERVATORY AAO/ERS_SPEC_4
DRAMA Software Report 4
Version 0.3

Tony Farrell
16-Mar-94

Drama Error reporting system (Ers)

Contents

1 Introduction

The purpose of the Drama Error Reporting System (**Ers**) is to provide facilities for constructing and storing reported error messages and for the delivery of those messages to the user via a technique appropriate to the program being run.

The major design issue in a Error Reporting System is that it allow low level packages to report errors using the appropriate output system for the application they are contained within, while ensuring that those packages can be written independently of the output systems of any application using them. For example, consider the Self Defining Data System (**SDS**). **SDS** may be used in standalone programs, such as one to list **SDS** structures in a file. In such applications, messages are normally just written to the standard output device (say using the C language routine `printf`). But when the **SDS** routines called in such an application wish to report an error, they cannot assume this is correct, since they could also be part of a **Dits** application where the correct user interface to send the message to is an X-windows application on another node in a network.

Ers allows low level packages to report errors in such a way that they can be sure it is sent to the correct user interface, while allowing the application to remain totally independent of the user interface.

2 Copyright details

Whilst most of this software is subject to the AAO copyright (commerical use requires AAT board approval), this product includes software developed by the University of California, Berkeley and its contributors.

3 Why not EMS/ERR

You may ask why another package is required to do this job, as Starlink already provides a similar facility in the EMS [2] and ERR [3] packages. The following reasons are given for writing yet another package

- EMS/ERR is largely written in Fortran. This makes it unsuitable for real-time systems for which Fortran is probably not available.

- EMS/ERR uses its own formating technique. This is expensive both in time and space. It is preferable to use the C `printf` style which although it may not be more efficient time wise, it is already present in most programs using C, so why waste memory implementing another technique.
- The interface to EMS/ERR is complicated by the requirements of backward compatiabilty with older systems and the use of Fortran.
- EMS/ERR provides no support for special attributes attached to messages to allow classes of messages to be differentated.
- EMS/ERR provides no inbuilt support for logging.
- EMS/ERR provides limited support for the ability to send muliple lines in one message, such as we can now do using SDS.

4 Usage

4.1 Overview

In a program or package consisting of many levels of subroutines, each routine which has something informative to say about an error should be able to contribute to the information 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 occured 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. In addition, 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.

4.1.1 Inherited status checking

The recommended method of indicating when errors have occurred 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 STATUS is input with a value other than 0.
- 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.

- In C, STATUS should be defined as a pointer to a `StatusType`, with the actual status being the value being pointed too. `StatusType` is defined in the `status.h` include file.

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

```
void routn( int value, StatusType * status)
{

/*
 *    Check the inherited global status.
 */
      if ( *status != 0 ) return;

      <application code>

}
```

If an error occurs within the "application code" of such a subroutine, then STATUS is set to a value which is not zero, 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. The only routines which do not require a status argument are functions with no other arguments, which cannot fail and which return a single value, say a funciton to fetch the current time.

4.2 Reporting errors

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

```
      ErsRep(flags, status, format, [arg ,  [...] );
```

Here, the flags argument in a bit mask which can influence the operation of the **Ers** system and are also passed to the logging and output systems (to be described later). Bits 0 to 7 are reserved to the **Ers** while bits bits 8 - 15 are available for logging systems (described later). The following masks are defined in `Ers.h` and may be `OR`ed together for the desired effect-

ERS_M_NOFMT Don't format the string. Any formating arguments are ignored and the format string is used exactly as specified.

ERS_M_HIGHLIGHT Suggest to the user interface that the message be highlighted in some way (say reverse video or a bold font).

ERS_M_BELL Suggest to the user interface that the terminal bell (or an equivalent) should be rung when the message is output

ERS_M_ALARM Suggest to the user interface that this is an urgent message which should be acknowledged by the user.

The `status` argument is the inherited status. `ErsRep` breaks the inherited status rules on two points. First the location of the argument, which is comes second not last. This is to allow a variable length argument list. Second, the routine works regardless of the value of the status argument.

The `format` argument and all following arguments are used to construct the message text. Formating is done using C run time library functions, see the description of the C `printf` function for details. You should not include special characters in the formats (e.g. `\n` , `\033` etc.) since you don't know what type of user interface your message is being sent to. Only characters for which the C funciton `isprint` returns true should be used [1].

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> )
{
    *status = myerrorcode;
    ErsRep(ERS_M_BELL,status,"Value of %d is invalid",value);
}
else
    ...
```

4.3 The contents 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 `ErsRep` 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]Messages requiring mulitiple lines should be output using multiple calls to `ErsRep`.

4.4 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. */
    smooth( nx, ny, sky, status );
    if ( *status != 0 )
        ErsRep(ERS_M_NOFMT,status,"SKYOFF: Failed to smooth sky values.");
    else
    {

/*      Smooth the object values. */
        smooth( nx, ny, object, status );
        if ( *status != 0 )
            ErsRep(ERS_M_NOFMT,status,"SKYOFF: Failed to smooth object values.");
        else
                ....
    }
```

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: Failed to smooth sky values.
```

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.

On a side issue, notice the use of the flag `ERS_M_NOFMT` in the above calls. We set this flag because we have not specified any formating arguments. We don't need to do this, since the formating routine can work this out, but it is more efficient to specify the flag as this lets `ErsRep` know the time comsuming formating can be avoided.

4.5 Deferred error reporting

Although the action of the subroutine `ErsRep` is to report an error to the Error System, 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 `ErsPush` , `ErsPop`, `ErsFlush` and `ErsAnnul`. This section describes the function of these subroutines and how they are used.

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:

```
void helper (char *line, StatusType *status)
{
    ...

/*
 *  Check for end-of-file error
 */
    if (feof(instream))
    {
        *status = <end-of-file error code>;
        ErsRep(ERS_M_NOFMT,status,"End of input file reached");
    }
/*
 *  Check for input error
 */
    else if (ferror(instream))
    {
        *status = <input error code>;
        ErsRep(0,status,"Error encountered during data input, errno = %x",errno);
    }
}
```

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 `ErsPush` before it calls HELPER. This has the effect of ensuring that all subsequent error messages are deferred by the Error System and stored in an "error table". `ErsPush` also starts a new "error context" which is independent of any previous error messages or message tokens. A return to the previous context can later be made by calling `ErsPop`, whereupon any messages in the new error context are transferred to the previous context. In this way, no existing error messages can be lost through the deferral mechanism. Calls to `ErsPush` and `ErsPop` should always occur in matching pairs and can be nested if required.

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

```
void helped (StatusType *status)
{

      ...

/*  Create a new error context. */
      ErsPush();

      /* any error messages from HELPER are now deferred */

      helper( line, status );

/*  Release the current error context. */

      ErsPop();

      ...
}
```

by calling `ErsPush` 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 `ErsAnnul(status)`, which "annuls" the error, deleting any deferred error messages in the current context and resetting `status` to zero. 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 `ErsRep` could also be used after `ErsAnnul` to replace the initial error condition with another more appropriate one, although this is not often done.)
- By calling `ErsFlush(status)`, which "flushes out" the error, sending any deferred error messages in the current context to the user and resetting `status` to zero. 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 `ErsFlush` before going on to process the next file.

When `ErsPop` is called, the new error context created by `ErsPush` ceases to exist and any error messages still remaining in it are transferred to the previous context.

Here is the previous example, elaborated to demonstrate the use of `ErsAnnul`. 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:

```
void helped (StatusType *status)
{
      int eof = 0;              /* end-of-file flag */
      ...

/*  Create a new error context. */
      ErsPush();

/*    any error messages from HELPER are now deferred */

      helper( line, status );

/*  Trap end-of-file  status and annul any report messages */

     if (*status == <end-of-file error status> )
     {
        ErsAnnul(status);
        eof = 1;
     }


/*  Release the current error context. */

      ErsPop();

      ...
}
```

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 `ErsPop`.

4.6 ErsOut

In some high level code, the sequence

```
    ErsRep(...);
    ErsFlush(...);
```

may be common. This sequence can be replaced by a single call to `ErsOut`, which has the same calling sequence as `ErsRep`. Note that `ErsOut`, like `ErsFlush`, removes the ability of higher level code to handle the error and thus the use of `ErsOut` should be avoided in libraries which may be used by other applications.

5 Error output

The final thing required is the actual ability to output the error. As mentioned above, errors are output by a call to `ErsFlush`. But, as this routine could be called by library routines which knows nothing about the user interface, just how do they get there.

The solution is in the use of `ErsStart` and `ErsStop`. There should be only one call to each of these routines in each application and they should be from a level high enough to know about the user interface in use. The format of `ErsStart` is

```
extern ErsTaskIdType ErsStart(
                ErsOutRoutineType outRoutine,
                void * outArg,
                ErsLogRoutineType logRoutine,
                void * logArg,
                StatusType * status);
```

Where-

- **outRoutine** is the actual output routine called whenever `ErsFlush` is called. Its format is-

  ```
  void (*ErsOutRoutineType)(
          void * outArg,
          unsigned int count,
          ErsMessageType messages[],
          StatusType * status);
  ```

  - **outArg** is the argument supplied to `ErsStart`.
  - **count** is the number of messages to be output.
  - **messages** is an array of the messages. The format is described below.

- **outArg** is an argument which is passed directly to `outRoutine`.
- **logRoutine** a logging routine called whenever `ErsRep` is called. Its format is-

  ```
  void (*ErsLogRoutineType)(
          void * logArg,
          ErsMessageType const * message,
          StatusType * status);
  ```

  - **logArg** is the argument supplied to `ErsStart`.
  - **message** is the message to log. The format is described below.

  If logging is not required, specify 0.

- **logArg** is an argument which is passed directly to `logRoutine`.

The type `ErsMessageType` is defined as follows

```
typedef struct {
        StatusType mesStatus;
        unsigned int context;
        int    flags;
        char   message[ERS_C_LEN];
            } ErsMessageType;
```

Where

- **mesStatus** is the status supplied in the call to `ErsRep`.
- **context** is the context level at the time of the call to `ErsRep`.
- **flags** is the flags arguments supplied in the call to `ErsRep`.

- **message** is the formated message text, as a null terminated string.

If zero is specified as the output routine, then the messages are written to the standard error output device using C run time library routines.

If `ErsStart` is not called before calls `ErsRep`, then the messages are output immediately to the standard error output device using C run time library routines. No defered error reporting occurs and the rest of the **Ers** routines have no effect. `ErsStop` reverts the system to this state if `ErsStart` has been called, after flushing any remaining messages.

The return value from `ErsStatus` is only of use in real time systems such as VxWorks. It is explained in section 5.2.

5.1 Logging

Sometimes it is nice to log all messages reported using `ErsRep`, even if they are later annulled. The `logRoutine` argument to `ErsStart` supports this. It is called for every call to `ErsRep`

5.2 The Task Id

Some systems, such as VxWorks based systems, run all programs in a common address space. In such systems, static and global variables can be seen by all tasks. To allow tasks to have private copies of static and global variables it is possible to have such variables saved and restored during task context switching. Ers uses this technique to store task specific information. In such systems, it is sometimes necessary to call Ers routines outside the context of a task (say in an interrupt handler routine), during which the task specific information will be unavailable. The routines `ErsEnableTask` and `ErsRestoreTask` allow this.

The technique requires the value returned from `ErsStart` to be available. You must supply this value as the first argument to a call to `ErsEnableTask`. In this example, the value is passed as the argument to the interrupt handler. (A type of ErsTaskIdType will fit in a `void *`).

```
int my_isr(void * parameter)
{
    ErsTaskIdType SavedId;
    ErsEnableTask((ErsTaskIdType)(parameter),&SavedId);

    /* Calls to ErsRep are now possible.  This allows me to call routines
       in an interrrupt handler which use ErsRep to report problems
     */

    ErsRestoreTask(SavedId);
}
```

Once Ers is enabled in the interrupt handler, the only call it makes sense to invoke is `ErsRep`, which will send the message to some appropriate device (No stacking occurs). Calls to `ErsPush`, `ErsPop` and `ErsAnnul` can be made but will be ignored. A call to `ErsFlush` or `ErsClear` is an error and will be reported as such.

6 Sprintf

Many C Run-time library routines which write to strings do not support any technique to determine the maximum length of the string which is being written. As a result it is easy to overwrite the stack. One such routine, `sprintf`, is requried by **Ers**. In order to avoid stack problems, a special version of this routine and an associated routine, `vsprintf`, were written. The routines `ErsSPrintf` and `ErsVSPrintf` provide an extra argument over `sprintf` and `vsprintf`. The extra argument is before the other arguments and indicates the maximum length of the output string.

I would like to suggest you use the **Ers** routines instead of the C Run-time library routines to avoid stack overwrite problems.

7 Use in Starlink programs

Some packages using **Ers** will probably be used in Starlink software environment programs (such as ADAM tasks). In this case, it would be nice if the **Ers** routines work nicely with the Starlink EMS/ERR routines. A special copy of the library provides this support. In this version, all the **Ers** routines except `ErsStart` and `ErsStop` are mapped to the appropriate Starlink routines.

8 Availability, Compiling and linking with Ers

Ers has been compiled under VAX/VMS C, the default compilers on Ultrix and SunOS, GNU C on Ultrix, SunOS and under VxWorks, Microsoft Quick C under MSDOS and the MPW compiler on a Macintosh. Under MPW, some modification may be required to replace calls to fprintf and fputs (file stderr) with an appropriate call for MPW.

Three include files are provided. `Ers.h` contians the function prototypes and defines the various **Ers** constants. The file `Ers_Err.h` contains the definitions of the **Ers** error codes, while the file `Ers_Err_msgt.h` contains the message table definition for using in calls to `MessPutFacility`.

We assume here the software organisation described in [1] for both VMS and UNIX machines.

8.1 Building under VMS

To build programs using **Ers** under VMS, you must first execute the `DRAMASTART` command.

Compilation time - include files

The **Ers** include files can now be found in `ERS_DIR:`. By using an appropriate command line to the C compiler[2], you can specify them using simple double quote notation, such as-

```
#include "Ers.h"
```

[2]See the `/INCLUDE=` qualifier to the VMS C compiler.

Link time

To use the normal **Ers** routines, link against the library `ERS_DIR:ERS.OLB`.

To use the Starlink version, you must execute the local Adam development startup. This is done with the following sequence-

```
@ssc:login
adamstart
adamdev
ladamstart
```

Link against `ERS_DIR:ERS_STAR.OLB` and `CNF_IMAGE/OPT`.

8.2 Building under UNIX

To build programs using **Ers** under VMS, you must first execute the local DRAMA development startup. This is done with the command `~drama/dramastart`.

Compilation time - include files

The **Ers** include files can now be found in the location referenced to by the environment variable ERS_DIR.

In your source code, you should specify them using simple double quote notation, such as-

```
#include "Ers.h"
```

Link time

The normal version of the normal **Ers** library can be found in $ERS_LIB/libers.a.

To link against the Starlink version, include '$ERS_LIB/ers_star_link' on the compiler command line. The quotes surrounding **$ERS_LIB/ers_star_link** are grave accents (ascii code 60 hex).

Note that you may need to do the normal Starlink startup - "`source /star/etc/login`", after doing the `dramastart` command, for this to work.

8.3 Building under VxWorks

To build programs using **Ers** under VMS, you must first execute the local DRAMA development startup on the development machine (The Sun). This is done with the command "`~drama/dramastart vw68k`" (For 680x0 based VxWorks machines).

Compilation time - include files

The Ers include files can now be found in the location referenced to by the environment variable ERS_DIR.

In your source code, you should specify them using simple double quote notation, such as-

```
#include "Ers.h"
```

Link time

The VxWorks version of the normal **Ers** library can be found in ERS_LIB/libers.a. There is no Starlink version available under VxWorks since Starlink itself is not available under VxWorks.

References

[1] Tony Farrell, AAO . *23-Dec-1992, DRAMA Software Organisation.* Anglo-Australian Observatory bf DRAMA Software Document 2.

[2] P C T Rees, Starlink Project. *8-Nov-1991, EMS Error Message Sevice Programmer's Manual.* Starlink User Note 4.3.

[3] P C T Rees, Starlink Project. *8-Nov-1991, MSG and ERR Message and Error Reporting Systems, Programmer's Manual.* Starlink User Note 104.3.

A Detailed Subroutine Descriptions

A.1 ErsAnnul — Annul all Error messages in the current context.

Function: Annul all Error messages in the current context.

Description: All pending messages at the current context are annulled, i.e. deleted. The context level does not change

Language: C

Call:
(Void) = ErsAnnul (status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(<) status (StatusType *) Set to zero.

Include files: Ers.h

External functions used: none

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO

A.2 ErsClear — Flush all error messages at all contexts and reset to base context

Function: Flush all error messages at all contexts and reset to base context

Description: All messages are written to the user. The context level does not change is reset to the base context.

Messages are written using the output routine supplied by the user when ErsStart was called. If no routine was supplied, then the messages are written to the standard Error output device using the C run time library.

Language: C

Call:
(Void) = ErsClear (status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(!) status (StatusType *) Set to `ERS__NOTACTIVE` if `ERS` is not active. Otherwise, as per ErsFlush. Unlike ErsFlush, this routine does not work if status is bad on entry.

Include files: Ers.h

External functions used:

ErsFlush	Ers	Flush error messages.
fprintf	Crtl	Write a message.

External values used: stderr (Crtl) The standard error device.

Prior requirements: ErsStart should have been called.

Support: Tony Farrell, `AAO`

A.3 ErsEnableTask — Enable Ers calls within an interrupt handler.

Function: Enable Ers calls within an interrupt handler.

Description: Some Systems, such as VxWorks based systems, run all programs in a common address space. In such systems static and global variables can be seen by all tasks. To allow tasks to have private copies of static and global variables it is possible to have such variables saved and restored during task context switching. Ers uses this technique to store task specific information. In such systems it is sometimes necessary to call Ers routines outside the context of a task (say in an interrupt routine), during which the task specific information will be unavailable. This routine is used in conjunction with ErsStart and ErsRestoreTask to make the task specific information available in such places.

This call is normally made at the begining of an interrupt handler. The argument should be a value previously returned by a call made to ErsStart() when executing in normal task context. After this call is made, Ers routines can be invoked although all that they do is report error using a method appropaite to the interrupt handler.

The value returned by this function should be supplied to ErsRestoreTask after before you exit the interrupt handler to ensure the task that was interrupted is restored to its original state.

This call is not neccessary or desirable on systems with process specific address spaces (`VMS` or `UNIX`). On such systems it does nothing.

Note that on VxWorks, if this function is called when not running at interrupt context, then taskLock() will be invoked to lock the current task as the running task - to avoid corruption issues.

Language: C

Call:
(Void) = ErsEnableTask (TaskId, SavedTaskId)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) TaskId (ErsTaskIdType) A value returned by DitsGetTaskID.

(<) SavedTaskId (ErsTaskIdType *) The value which is to be passed to ErsRestoreTask is put here.

Include files: Ers.h

External functions used: None

External values used: None

Prior requirements: ErsStart should have been called.

Support: Tony Farrell, `AAO`

A.4 ErsFlush — Flush all error messages at the current context.

Function: Flush all error messages at the current context.

Description: All messages at the current context are written to the user. The context level does not change.

Messages are written using the output routine supplied by the user when ErsStart was called. If no routine was supplied, then the messages are written to the standard Error output device using the C run time library.

Language: C

Call:
(Void) = ErsFlush (status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(!) status (StatusType *) Set to zero, unless an output error occurs, in which case it is the error code returned by the output routine.

Include files: Ers.h

External functions used:

fprintf	Crtl	Formated print.
fputs	Crtl	Output a string
logMsg	VxWorks	Log a message

External values used: stderr (Crtl) The standard error device.

Prior requirements: none

Support: Tony Farrell, AAO

A.5 ErsGetAtCtx — Access all ERS messages reported at the current contact.

Function: Access all ERS messages reported at the current contact.

Description:

Language: C

Call:
(Void) = ErsCtx (status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

- **(<) count (unsigned int *)** Number of message at the current context.
- **(<) messageArray (ErsMessageType **)** Will be set to an array of ErsMessageType of size at least "count". If count is set to zero, then this may be set to a null pointer.
- **(<) status (StatusType *)** Set to zero.

Include files: Ers.h

External functions used: none

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO

A.6 ErsGetTaskId — Get ERS task id for use with ErsEnableTask.

Function: Get ERS task id for use with ErsEnableTask.

Description: Some Systems, such as VxWorks based systems, run all programs in a common address space. In such systems static and global variables can be seen by all tasks. To allow tasks to have private copies of static and global variables it is possible to have such variables saved and restored during task context switching. Ers uses this technique to store task specific information. In such systems it is sometimes necessary to call Ers routines outside the context of a task (say in an interrupt routine), during which the task specific information will be unavailable. This routine is used in conjunction with ErsEnableTask and ErsRestoreTask to make the task specific information available in such places.

This routine can be used where the task ID returned by ErsStart(3) is not avaiable. It should be invoked in non-interrupt handler code to fetch the task id for passing to ErsEnableTask(3).

Language: C

Call:
(ErsTaskIdType) = ErsGetTaskId (StatusType *status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(!) status (StatusType *) Modified status.

Include files: Ers.h

External functions used: None

External values used: None

Function Value: A value which can be passed to ErsEnableTask from an interrupt handler. If status is non-zero, zero is returned.

Prior requirements: ErsStart should have been called.

Support: Tony Farrell, AAO

A.7 ErsOut — Report an Error message.

Function: Report an Error message.

Description: Implements the equivalent of a call to ErsRep followed by a call to ErsFlush.

The format and it's arguments are the the same as used by the `printf` C `RTL` function.

Language: C

Call:
(Void) = ErsOut (flags, status, format, [arg ,[...]])

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) flags (Int) Message flags. These flags influence the operation of the Ers system and are also passed to the logging and output routines when these are called for this message. Bits 0 to 7 are reserved to the Ers system while bits 8 - 15 are available for logging systems. Other bits should not be used since they may not be available on 16bit machines. The following masks are defined and may be ORed for the desired effect-

`ERS_M_NOFMT`	Don't format the string. Any formating arguments are ignored and the format string is used as specified. Deprecated!! Call ErsRepNF() followed by ErsFlush() instead of specifying this flag to

() ErsOut () to avoid compiler warnings.

`ERS_M_HIGHLIGHT`	Suggest to the user interface that the message should be highlighted.
`ERS_M_BELL`	Suggest to the user interface that the terminal bell (or an equivalent) should be rung when the message is output.
`ERS_M_ALARM`	Suggest to the user interface that this is urgent message which should be acknowledged by the user.

(!) status (StatusType *) Set to zero unless the message table is full, when status is set to `ERS__NOSPACE`, or an output error occurs, in which case it is set to the error code returned by the output routine.

(>) format (Char *) A format statement. See the description of the C printf function.

(>) arg (assorted) Formating arguments. Set the description of the C printf function.

Include files: Ers.h

External functions used:

va_start	Crtl	Start a variable argument session.
va_args	Crtl	Get an argument.
va_end	Crtl	End a variable argument session.
fputs	Crtl	Output a string.
fprint	Crtl	Output a formated string.
memcpy	Crtl	Copy one string to another.
logMsg	VxWorks	Log a message.
ErsVSPrintf	Ers	Safe format.
ErsFlush	Ers	Flush error messages.

External values used: stderr (Crtl) The standard error device.

Prior requirements: none

Support: Tony Farrell, AAO

A.8 ErsPop — Decrease Error context level

Function: Decrease Error context level

Description: Pops the Error message table context, returning the Error Message Service to the previous context. Note that any messages pending output will be passed to this previous context, not annulled.

Language: C

Call:
(Void) = ErsPop ()

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

Include files: Ers.h

External functions used: none

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO

A.9 ErsPush — Increase Error context level

Function: Increase Error context level

Description: Begin a new Error reporting context so that delivery of subsequently reported Error messages is defered and the messages held in the Error table. A Subsequent call to `ErsAnnul` or `ErsFlush` will only annul or flush the contexts of the Error table within this new context.

Language: C

Call:
(Void) = ErsPush ()

Parameters: (“>” input, “!” modified, “W” workspace, “<” output) none

Include files: Ers.h

External functions used: none

External values used: none

Prior requirements: none

Support: Tony Farrell, `AAO`

A.10 ErsRep — Report an Error message.

Function: Report an Error message.

Description: 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 successfull completion, status is returned unchanged unless an Error occurs in this routine.

Output will only occur if the context is 0, in which case ErsStart has not been called and the message will be output using `fputs` to `stderr`.

If a logging routine was specified when ErsStart was called, then the logging routine is called after the message is formated.

The format and it's arguments are the the same as used by the `printf` C `RTL` function.

The maximum length of stored messages is given by the macro `ERS_C_LEN` (currently 200 characters including the null terminator) so the result of the format should be less then this value. If the format results in a string greater then this value it is truncated.

Language: C

Call:
(Void) = ErsRep (flags, status, format, [arg ,[...]])

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) **flags (Int)** Message flags. These flags influence the operation of the Ers system and are also passed to the logging and output routines when these are called for this message. Bits 0 to 7 are reserved to the Ers system while bits 8 - 15 are available for logging systems. Other bits should not be used since they may not be available on 16bit machines. The following masks are defined and may be ORed for the desired effect-

`ERS_M_NOFMT`	Don't format the string. Any formating arguments are ignored and the format string is used as specified. Deprecated!! Call ErsRepNF() instead of specifying this flag to ErsRep() to avoid compiler warnings.
`ERS_M_HIGHLIGHT`	Suggest to the user interface that the message should be highlighted.
`ERS_M_BELL`	Suggest to the user interface that the terminal bell (or an equivalent) should be rung when the message is output.
`ERS_M_ALARM`	Suggest to the user interface that this is urgent message which should be acknowledged by the user.

(!) **status (StatusType *)** The routine works regardless of the value of status. If the message table is full, then status is set to `ERS__NOSPACE`, otherwise, it is not touched.

(>) **format (Char *)** A format statement. See the description of the C printf function.

(>) **arg (assorted)** Formating arguments. Set the description of the C printf function.

Include files: Ers.h

External functions used:

va_start	Crtl	Start a variable argument session.
va_args	Crtl	Get an argument.
va_end	Crtl	End a variable argument session.
fputs	Crtl	Output a string.
fprint	Crtl	Output a formated string.
memcpy	Crtl	Copy one string to another.
logMsg	VxWorks	Log a message.
ErsVSPrintf	Ers	Safe format.

External values used: stderr (Crtl) The standard error device.

Prior requirements: none

Support: Tony Farrell, AAO

A.11 ErsRepNF — Report an Error message. No formating.

Function: Report an Error message. No formating.

Description: 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 successfull completion, status is returned unchanged unless an Error occurs in this routine.

Output will only occur if the context is 0, in which case ErsStart has not been called and the message will be output using `fputs` to `stderr`.

If a logging routine was specified when ErsStart was called, then the logging routine is called after the message is formated.

The format and it's arguments are the the same as used by the `printf` C `RTL` function.

The maximum length of stored messages is given by the macro `ERS_C_LEN` (currently 200 characters including the null terminator) so the result of the format should be less then this value. If the format results in a string greater then this value it is truncated.

Language: C

Call:
(Void) = ErsRepNF (flags, status, string)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) **flags (Int)** Message flags. These flags influence the operation of the Ers system and are also passed to the logging and output routines when these are called for this message. Bits 0 to 7 are reserved to the Ers system while bits 8 - 15 are available for logging systems. Other bits should not be used since they may not be available on 16bit machines. The following masks are defined and may be ORed for the desired effect-

`ERS_M_NOFMT`	Ignored.
`ERS_M_HIGHLIGHT`	Suggest to the user interface that the message should be highlighted.
`ERS_M_BELL`	Suggest to the user interface that the terminal bell (or an equivalent) should be rung when the message is output.
`ERS_M_ALARM`	Suggest to the user interface that this is urgent message which should be acknowledged by the user.

(!) **status (StatusType *)** The routine works regardless of the value of status. If the message table is full, then status is set to `ERS__NOSPACE`, otherwise, it is not touched.

(>) **string (Char *)** String to output.

Include files: Ers.h

External functions used:

fputs	Crtl	Output a string.
fprint	Crtl	Output a formated string.
memcpy	Crtl	Copy one string to another.
logMsg	VxWorks	Log a message.
ErsVSPrintf	Ers	Safe format.

External values used: stderr (Crtl) The standard error device.

Prior requirements: none

Support: Tony Farrell, AAO

A.12 ErsRestoreTask — Restore the interrupted Task Id

Function: Restore the interrupted Task Id

Description: Some Systems, such as VxWorks based systems, run all programs in a common address space. In such systems static and global variables can be seen by all tasks. To allow tasks to have private copies of static and global variables it is possible to have such variables saved and restored during task context switching. Ers uses this technique to store tErsGetAtCtxask specific information. In such systems it is sometimes necessary to call Ers routines outside the context of a task (say in an interrupt routine), during which the task specific information will be unavailable. This routine is used in conjunction with ErsStart and ErsEnableTask to make the task specific information available in such places.

This call is made in interrupt handlers. The argument should be the value returned by a previous call to ErsEnableTask in the interrupt handler. After this call is made, Ers routines can no longer be used.

This call is not neccessary or desirable on systems with process specific address spaces (`VMS` or `UNIX`). On such systems it does nothing.

Language: C

Call:
(Void) = ErsRestoreTask (TaskId)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) TaskId (ErsTaskIdType) A value returned by ErsEnableTask.

Include files: Ers.h

External functions used: None

External values used: None

Prior requirements: ErsStart should have been called.

Support: Tony Farrell, `AAO`

A.13 ErsSPrintf — A safe version of the C RTL sprintf function.

Function: A safe version of the C RTL sprintf function.

Description: The standard C RTL version of sprintf is unsafe as nothing limits the length of the output string. It is easy to overwrite the stack. By providing a length argument string argument, this routine implements a safe version of sprintf.

See ErsVSPrintf() for more details.

Language: C

Call:
(int) = ErsSPrintf(length, string, format, args...)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) length (int) The length of string.

(<) string (char *) The pointer to the output string

(>) format (char *) A format specification (>) arg... (anything) argument list

Function value: EOF indicates the format string exceeds the length available, otherwise, the number of characters output.

Include files: Ers.h, stdio.h

External functions used:

ErsVSPrintf	Ers	A save version of vsprintf.

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO

A.14 ErsSetLogRoutine — Change the Ers log routine.

Function: Change the Ers log routine.

Description: Allows us to change the routine used to log `ERS` messages.

Language: C

Call:
(void) = ErsSetLogRoutine (logRoutine, logArg, oldLogRoutine, oldLogArg, status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

- **(>) logRoutine (ErsLogRoutineType)** If non zero, this routine will be called to log any messages reported by ErsRep.
- **(>) logArg (void *)** Passed directly to logRoutine as its first argument and not examined by Ers.
- **(>) oldLogRoutine (ErsLogRoutineType *)** The previous log routine is returned here - if non-zero, the new routine should call this routine after it has done its job.
- **(>) oldLogArg (void *)** Should be passed as the first argument to the old log routine.
- **(!) status (StatusType *)** Modified status.

Include files: Ers.h

External functions used: none

External values used: none

Prior requirements: none

Support: Tony Farrell, `AAO`

A.15 ErsStart — Startup Error reporting system.

Function: Startup Error reporting system.

Description: Startup the Error reporting system. Calls may be made to the other Ers routines without calling this routine, but if this is done, then messages reported with ErsRep are written directly to the user standard Error output, not stored.

Language: C

Call:
(ErsTaskIdType) = ErsStart (outRoutine, outArg, logRoutine, logArg, status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

- **(>) outRoutine (ErsOutRoutineType)** If non zero, this routine will be called to output any messages output by ErsFlush.
- **(>) outArg (void *)** Passed directly to outRoutine as its first argument and not examined by Ers.
- **(>) logRoutine (ErsLogRoutineType)** If non zero, this routine will be called to log any messages reported by ErsRep.
- **(>) logArg (void *)** Passed directly to logRoutine as its first argument and not examined by Ers.
- **(!) status (StatusType *)** Modified status. routine returns immediately if non-zero. If Ers is already active, it is set to `ERS__ACTIVE`. An error allocating space for the table will set status to `ERS__MALLOCERR`. Under VxWorks, an error setting up a task varable will cause it to be set to `ERS__TASKVARERR`.

Include files: Ers.h

Function Value: A value which can be passed to ErsEnableTask from an interrupt handler.

External functions used:

malloc	Crtl	Allocate memory.
taskVarAdd	Crtl	Add a task variable. none

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO

A.16 ErsStop — Shutdown Error reporting system.

Function: Shutdown Error reporting system.

Description: The major effect of this routine is that in any future calls to ErsRep, the message is reported immediately to stderr. The Output and logging routine supplied by the previous call to ErsStart are forgotton.

The first thing this routine does is flush all pending messages.

Language: C

Call:
(Void) = ErsStop (status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(!) status (StatusType *) Modified status. routine returns immediately if non-zero. If Ers is not active, it is set to `ERS__NOTACTIVE`. Under VxWorks, an error deleting a task varable will cause it to be set to `ERS__TASKVARERR`.

Include files: Ers.h

External functions used:

ErsClear	Ers	Flush all outstanding errors.
free	Crtl	Release memory.
taskVarDelete	VxWorks	Delete a task variable.

External values used: none

Prior requirements: none

Support: Tony Farrell, `AAO`

A.17 ErsVSPrintf — A safe version of the C RTL vsprintf function.

Function: A safe version of the C RTL vsprintf function.

Description: The standard C RTL version of vsprintf is unsafe as nothing limits the length of the output string. It is easy to overwrite the stack. By providing a length argument string argument, this routine implements a safe version of vsprintf.

When not under VxWorks, this uses vsnprintf().

Under VxWorks, the fioFormatV routine is used.

Language: C

Call:

(int) = ErsVSPrintf(length, string, format, arg)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

- **(>) length (int)** The length of string.
- **(<) string (char *)** The pointer to the output string
- **(>) format (char *)** A format specification
- **(>) arg (va_list)** Variable argument list

Function value: EOF indicates the format string exceeds the length available, otherwise, the number of characters output.

Include files: Ers.h, stdio.h

External functions used:

fioFormatV	VxWorks	Do a C style format.

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO