# The SL Error System Software User Manual

*Frederick G.M. Roeber*

ABSTRACT

The SL Control System consists of cooperating software packages running on computers distributed throughout the accelerator complex. There are many sources of error throughout this network of software, and the complexity of the interactions often makes it difficult to understand the implications of these errors. When a control action fails because of one or more errors, proper and efficient handling of the situation requires that precise, detailed reports, together with a presentation of the causes and effects of the errors, be presented to the agent responsible for the action. The SL Error System provides an easily used multilingual error reporting facility. Using it, software can easily generate detailed error reports in a way which fits its natural command/response operation. The System automatically records the chain of cause and effect, and reports everything to the responsible agent.

# Table of Contents

# Document Status Sheet

| Issue | Revision | Date | Reason for change |
|---|---|---|---|
| 1 | 0 | 20 October 1993 | First issue |
| 1 | 1 | 20 December 1993 | Minor corrections |
| 1 | 2 | 18 Jaunary 1994 | Glossary and Index |

This document (version 1.2) basically corresponds to the SL Error System as conceived and written by the original author, plus early feedback from users. At the time of writing, the System is undergoing some changes; including, but not limited to, the removal of the internationalization (multilingual) facilities. Also, the interaction with the SL RPC system will be changed in the near future. When more is known about these changes, this document should be amended.

This document was written following the guidelines of ESA standard PSS–05–05 Issue 1 (May 1992) “The Software User Manual.”

# Document Change Records

Version 1.0: Original Version (20 October 1993)

Version 1.1: Minor corrections (20 December 1993)

Version 1.2: Addition of glossary and index (18 January 1994)

# 1 Introduction

## 1.1 Intended Readership

This Software User Manual contains three major sections. The first one is an overview of the System, which includes a description of the philosophy behind this system and a general description of the features it provides. The second section is an instruction manual designed to illustrate the use of the System to people unfamiliar with it. The final section is the authoritative reference manual. There are two general categories of intended readers: the SL operations people and the developers of SL software. The SL operations people may need a general knowledge of the Error System, to best understand the errors presented to them. They should read the overview section. The software developers will need not only a general understanding, but instruction in the System's use, and a reference manual. They should read this entire manual.

## 1.2 Applicability Statement

At the time of writing of this document, the SL Error System was not under a rigorous revision control system, so it is impossible to state exactly to which version it applies. This initial document version was written by the System's original author, and basically corresponds to the system as he designed it. Elements known not to be resolved are so mentioned in the text. Also, at the time of writing, another developer has been assigned responsibility for the system and is amending the System off–line. Although the author has some idea of the intended changes (e.g. the removal of the Internationalization facilities), he has not attempted to document them here. This document should be amended when these changes have been completed.

## 1.3 Purpose

The SL Error System provides an error reporting facility for SL software. Using it, software can easily make detailed descriptions of error situations and report them to the agent responsible for the failed action. The System builds a description of the cause–and–effect chain of events involved in an error, for a better understanding of the origin and implications of the error. The final goal is the clear explanation for any error of what went wrong and why, to provide operators and other users sufficient information to determine how to resolve the problem.

To be useful, the System must be understood both by the SL operators, who will be presented with the error reports, and by the SL software developers, who will need to use it. This Software User Manual is intended to provide a general description of the philosophy and operation of the System for all users, and a detailed guide and reference for the developers.

## 1.4 How to use this document

There are three basic sections in this document: an overview (section 2), an instruction manual (section 3), and a reference manual (section 4). The overview section describes the purpose, basic principles, and elements of the SL Error System, as well as a summary of the developer responsibilities. This section should be read by all users. The instruction manual introduces the elements of the System, and should be read by developers new to the System. The reference manual gives a detailed description of everything, and should be consulted by developers as needed. It can also be read through by those people who read manuals. In addition, there are five appendices, which give the SL Error System errors, a list of acronyms and a glossary of terms used in this document, an index to this document, example program files, and information about using this System from *SLAPS*. They should be consulted as needed. Also, there is an "overhead" section, which gives basic information about this document itself (section 1). It is intended to be read by pedants.

## 1.5 Related documents

*Accessing Equipment in the SPS–LEP Controls Infrastructure: the "SL–EQUIP" Package,* Charrue P., SL/Note 93–86 (CO).

*An Improved Error Handling Mechanism for the New SPS Application Software,* Loughran S., SPS/AOP/OS/Note 46.

*The LEP Alarm System,* Tyrrell, M.W., ICALEPCS'91.

*New Control Architecture for the SPS Accelerator at CERN,* Kissler K.H. and Rausch R., CERN SL/91–(CO).

## 1.6 Conventions

In the overview section, terms with a specific meaning are italicised the first time they are used. In the instruction section, examples of source code are given in a fixed–width, typewriter–like "computer" font. Routines mentioned in the text are also given in this font. Variables and constants mentioned in the text are italicised. In the reference guide, code (including constants) is presented in "computer" font. References to standard on–line UNIX manual pages are also present in this font, in the usual form.

## 1.7 Problem reporting instructions

Comments, questions, and problem reports regarding the SL Error System should be sent via e–mail to either Frederick Roeber `<roeber@cern.ch>` (the original author), or Ann Sweeney `<sweeney@cern.ch>` (the current developer).

# 2 Overview

## 2.1 Purpose of the SL Error System

### 2.1.1 What is an error?

An *error* is the result of software not being able to perform its task. For example, software may return an error because

1. a necessary subroutine failed in a way from which it cannot be recovered,
2. an internal consistency check failed, or
3. an input is invalid or inconsistent.

An error is a report that the software cannot function properly.

There is often confusion between the roles of errors and *fault states*. (Fault states are also known as *alarms*.) A fault state can be in one of two states: it can be *active* or *not active*. An active fault state indicates that there is an existing situation that needs to be resolved. An active fault state may for example indicate that some piece of hardware or software is not operating properly. A fault state may also be activated by a properly working detector noticing a problem, for instance smoke or flooding. An error is a report that at some time a specific action failed.

### 2.1.2 To whom should errors be reported?

When the user of an application requests that some action be performed, any errors resulting from this action should be reported to this user as the result of his request.

When an error results from an automatic, ‘‘background’’ task, it is less clear to where the error should be reported. Since some automatic tasks are vital to machine operations (for example, many logging tasks), it was decided that such background task errors should be displayed on a general error screen in the control room (analogous to the alarm console). With appropriate software managing and filtering of these error reports, operations people can be informed when important automatic actions fail.

### 2.1.3 What does the SL Error System do?

The SL Error System reports errors to the ‘‘proper’’ place. An *error object* is created when software first detects a problem. If the action causing this error involved multiple layers of software, this error object is passed back up through the layers, collecting data as it goes. At the ‘‘top,’’ it is delivered to an *Error Server.*

An *Error Server* is an application program which runs on all operations consoles. One of these consoles is designated the *General Error Server.* When an Error Server receives an error object, it displays the associated information. Errors resulting from actions taken on an operations console are reported on the same console; all others are reported on the General Error Server.

## 2.2 Fundamental principles

### 2.2.1 The SL Control System environment

The SL Control System is composed of software which runs on hundreds of computers distributed throughout the accelerator complex. At one end, specialized equipment control software runs on the computers connected directly to the hardware equipment. At the other end, control–room workstations run the application programs used by the operators. Other workstations take care of background tasks, such as logging, or provide basic services, such as the databases. These computers are all connected together with a variety of networks.

Most control actions therefore involve at least two programs (at the equipment and workstation levels), and often many more. Complex operations can involve the coordination of scores of programs and the resources of many servers.

Some programs, particularly those running on the operations workstations, have considerable internal structure, and use a multitude of standard SL libraries.

*(For more information, see* SL/91–(CO).*)*

### 2.2.2 Error reporting history

Throughout the brief history of the SL Control System, a variety of error handling and reporting methods has been used.

The simplest of these methods is the typical UNIX approach of printing a brief message to `stderr` and exiting with some value. This has the advantage of simplicity, but no other. While this may be tolerable for "quick–hack" user interface applications, it is not recommended, particularly for remote servers.

A related approach used by some servers is to print a message to `stderr` (usually redirected to a log file, or more often `/dev/null`) and return a small integer to the client. Unfortunately, the small integers are usually chosen haphazardly, and do not uniquely identify the source of the problem. The log files (if any) are usually obscurely placed, too.

Most standard SL libraries also have a set of small integer error values, which also are in no way coordinated. Some packages, which call other software, make a practice of returning any error values they receive in the same way they return their own errors. This can lead to ambiguity.

Some packages make use of external variables to store additional error data; usually a brief string.

Software using the SPS Error System can send an integer, plus some number of limited–length strings, to an SPS Error Server previously *imported.* Much SPS software returns limited–length strings, with the single character 'E' return value to indicate an error.

The first version of the SL RPC system used a simple exception mechanism to report errors. Unfortunately, this mechanism was little understood and easily misused. Improper use was most often detected by the spectacular and confusing program crashes that quickly became associated with "The RPC System." The RPC developers quickly abandoned this method, and reverted to a magic return value. Unfortunately, this value can also lead to ambiguity.

The ARCH system was designed to be the error handling system for LEP. In this system, a unique identifier was generated any time an error occurred in the control system. This identifier was used as the return value for all subroutine and RPC calls, and all software associated with the action made RPC calls to a central server. This central server correlated and logged all of this information. Unfortunately, however, the system was felt to be rather baroque and complex, and it was little used.

The immediate forerunner of the SL Error System was the system used in the SL Measurement System, the SL Logging System, and the Database Monitor. This error system, written by this author and partly motivated by the error reporting facilities in the VMS and Apollo Domain operating systems, was a testbed for many SL Error System concepts. It has now been replaced by the SL Error System.

## 2.2.3 SL Error System concepts

### 2.2.3.1 Error Numbers

Every error of every program that uses the SL Error System has a unique number, called its *error number*. This number is a 32–bit integer. The most significant sixteen bits (the "high word") identify a particular *product*, which usually corresponds to a *SLAPS* product. The least significant sixteen bits (the "low word") identify a particular error within a product. The special case where the low word is zero is called the *error base* of the particular product, and is not used to indicate an error. The error bases are assigned by the *SLAPS administrator*. The special error base equal to zero is reserved.[1] The error number zero, usually referred to by the preprocessor macro *error_ok*, is used to indicate success (i.e. the lack of an error).

### 2.2.3.2 Error Messages

An *error message* is the set of data from a single layer of software with an error. It includes the error number, the filename of the program source file and the line number at which the message was reported, a text message describing the error in the default language of the software (usually English), and any parameters to be used by the text message.

### 2.2.3.3 Error Stacks

An *error stack* is an ordered set of error messages. Error messages are never seen alone, they are always handled in error stacks. However, an error stack may contain only one message. Error stacks are sometimes referred to as *error objects* by object–oriented people.

Error stacks are used to represent the chain of cause and effect involved in an action resulting in an error. This is best illustrated with a traditional poem:

*For the want of a nail, the shoe was lost,*
*For the want of the shoe, the horse was lost,*
*For the want of the horse, the rider was lost,*
*For the want of the rider, the message was lost,*
*For the want of the message, the battle was lost,*
*For the want of the battle, the war was lost,*
*For the want of the war, the kingdom was lost:*
*The kingdom was lost, all for the want of a horseshoe nail.*

Imagine a military historian studying such a situation. If he were presented with merely the topmost report, "The kingdom was lost," he would probably reply, "Yes I know, but why?" Presented with the lowest–level message, "Could not find a horseshoe nail," the response "so what?" springs to mind. Only when the entire stack of information is presented could he realize the significance of that battle, and the effect of communications breakdown.

### 2.2.3.4 Error Library

There is a library in the SLAPS repository, available in the *SLAPS* environment with `$SL_ERRORLIB`, which contains all of the Error System routines needed by software using the System. The corresponding header file, containing all needed types, macros, and prototypes, is `error_handler.h`. All types, macros, and routines are prefixed with "`error_`."

### 2.2.3.5 Exceptions

There is an exception facility available as part of the SL Error System. It may be used independently of the rest of the Error System, and the Error System may be used by programs

1. The error base `0` is used by the exception system, described later on. As the SL Error System internally uses the exception system, the exception system cannot use the Error System. Therefore, there are no errors with a base of zero.

which do not use the exception facility (though the Error Library uses it internally). However, as will be explained more fully in section 3, the exception facility and Error System complement each other very nicely.

This exception facility is implemented with preprocessor macros and library calls. It is based on the standard `sigsetjmp`/`siglongjmp` calls, and works on all SL platforms. The library is available in the SLAPS environment with `$SL_EXCEPTIONLIB`, and the include file is `exception.h`. This exception system does not suffer from the same problems that damned the RPC version 1 exception system.

#### 2.2.3.6 Internationalization

The SL Error System allows multilingual error messages. In addition to the default text provided in the error message, products may have *message catalogs* for specifying the error text in other languages. As the Error System only uses one of the available sets of strings in the catalogs, the others are available for other use by the product software. The error library has a routine which, given a product's error base, finds the proper message catalog in the SLAPS/SLOPS repository.

#### 2.2.3.7 Error Server

The error server is an application program that runs on all control room operations workstations. It receives error objects from programs running on the same computer, and displays the data. There is one particular control room workstation, the General Error Server, which also receives error objects from any programs running on computers without error servers.

The current error server is merely the SPS Error Server, although a new one, with full functionality, was foreseen. For now, refer to SPS/AOP/OS/Note 46.

## 2.3 Programmer responsibilities

### 2.3.1 Error Numbers

Software developers who wish to use the SL Error System in a product must obtain an error base from the SLAPS Administrator. Usually there is a one–to–one correspondance between SLAPS products and error bases, though this is just a general guideline. A preprocessor `#define` directive will be added to the SLAPS `error_bases.h` file, to define a macro for this base. This macro will be of the form `ERROR_`*product*`_BASE`.

The developer defines error numbers based on this value. Usually these will be *base+1, base+2, base+3,* etc., though systems with an already defined set of positive integer error numbers may be easily mapped into the SL Error System by merely biasing the numbers with the assigned base. If the product is or contains a *public library*, or if it is a server, the product's *public include file* should contain `#define` preprocessor directives defining mnemonic macros for these numbers. It is recommended that these be of the form `ERROR_`*product_error*, where the last part is somewhat descriptive of the problem.

### 2.3.2 Message Catalog

Developers who wish to write *internationalized* software, or who merely want to provide error messages in more than one language, may create standard UNIX–style message catalogs. There should be one catalog per language, for each error base. The first message set in each message catalog is used by the Error System. For each error message, the low word of the error number is used as the message number for the corresponding text.

The standard languages used in the SL Control System are "`en_US.iso88591`" and "`fr_FR.iso88591`," i.e., American English and French as spoken in France, both using the character set defined in ISO standard 8859–1. Any other language may be provided as well, though those supported by the control room workstations will be most useful. Please note that the software does not have to run on the control room workstations to use this multilingual facility. In fact, even software written for computers without internationalization support (e.g., *LynxOS ECAs*) may have error messages in multiple languages.

If a developer takes advantage of the message catalogs, the default error text hard–coded into the application may be in any language the developer chooses. If a developer does not provide message catalogs, the default messages should be in English.

### 2.3.3 Building Error Stacks

Every logical unit of software should have a unique error to report its failure. Usually, this unit is the function or procedure. When a unit receives an error from which it cannot recover, it should stack on its own error message before passing the error on.

### 2.3.4 Reporting errors

Every unit should report any error to wherever it received the request for the failed action. For subroutines, this usually means the error is returned to the caller; for servers, this means the error should be sent back to the client. At the very top, where there is no further software, the `error_output` routine should be used to report the error to an error server.

# 3 How to use the SL Error System

## 3.1 Building error stacks

The first example uses a C routine to calculate the formula

$$\frac{arcsin(a/b)}{\sqrt{1 - v^2}} .$$

Figure 1 contains the top–level function; figure 2 contains some subroutines.

```
void
F
(
    double a,
    double b,
    double v,
    double *pr
)
{
    double w, x, y, z;

    divide(a, b, &w);
    arcsin(w, &x);
    y = (1 - v*v);
    root(y, &z);

    divide(x, z, pr);
}
```

*Figure #1*

The problem with this example is that there are input values for which the function is not defined. The code shown blithely tries to calculate the result anyway, which can at best lead to a nonsensical value and at worst can terminate the program.

Specifically, the three problems are,

! The divisors may be zero,

! The argument of the arcsine may be out of range, or

! The argument of the square root may be negative.

Any of these problems may prevent the function from being calculated. (Also, there are range problems involved with very large numbers; for simplicity they are not treated here.)

```
void
divide
(
    double dividend,
    double divisor,
    double *pquotient
)
{
    *pquotient = dividend/divisor;
}

void
arcsin
(
    double x,
    double *py
)
{
    *py = asin(x);
}

void
root
(
    double s,
    double *pr
)
{
    *pr = sqrt(s);
}
```

*Figure #2*

Figure 3 shows versions of the subroutines which do error checking, and which return error stacks. Since each of the error cases involves a newly detected problem, each creates an error stack by passing `NULL` as the first parameter. The product header file, "`example.h`," is shown in figure 4.

The stacking of a second error message on an error stack is shown in figure 5, the main calculation function "`F`." At every place where it receives an error stack from a subroutine, it checks the value against *error_ok*. If it receives a non–empty error stack, it layers its own message on (after the "`recover`" label) and returns the two–level error stack.

```
error_t
divide
(
    double dividend,
    double divisor,
    double *pquotient
)
{
    if( fabs(divisor) < EPSILON )
    {
        return error_stack(NULL, ERROR_EXAMPLE_DIV_BY_ZERO,
            "Cannot divide %lg by %lg: divisor is too small.",
            dividend, divisor);
    }

    *pquotient = divident/divisor;
    return error_ok;
}

error_t
arcsin
(
    double x,
    double *py
)
{
    if( x < -1.0 || x > 1.0 )
    {
        return error_stack(NULL, ERROR_EXAMPLE_ASIN_RANGE,
            "Cannot take arcsin of %lg: value out of range.", x);
    }

    *py = asin(x);
    return error_ok;
}

void
root
(
    double s,
    double *pr
)
{
    if( s < 0.0 )
    {
        return error_stack(NULL, ERROR_EXAMPLE_SQRT_NEGATIVE,
            "Cannot take square root of %lg: argument is negative.", s);
    }

    *pr = sqrt(s);
    return error_ok;
}
```

*Figure #3*

```
#include "error_bases.h"     /* For ERROR_EXAMPLE_BASE */

#define ERROR_EXAMPLE_DIV_BY_ZERO    (ERROR_EXAMPLE_BASE + 1)
#define ERROR_EXAMPLE_ASIN_RANGE     (ERROR_EXAMPLE_BASE + 2)
#define ERROR_EXAMPLE_SQRT_NEGATIVE  (ERROR_EXAMPLE_BASE + 3)
#define ERROR_EXAMPLE_F              (ERROR_EXAMPLE_BASE + 4)
```

*Figure #4:* `example.h`

```
#include <math.h>
#include "error_handler.h"
#include "a.h"


error_t
F
(
    double a,
    double b,
    double v,
    double *pr
)
{
    double w, x, y, z;
    error_t e;

    e = divide(a, b, &w);   if( e != error_ok ) goto recover;
    e = arcsin(w, &x);      if( e != error_ok ) goto recover;
    y = (1 - v*v);
    e = root(y, &z);        if( e != error_ok ) goto recover;

    e = divide(x, z, pr);   if( e != error_ok ) goto recover;

    return error_ok;

  recover:
    return error_stack(e, ERROR_EXAMPLE_F,
        "Cannot calculate F(%lg, %lg, %lg).", a, b, v);
}
```

*Figure #5*

Two message catalogs for this example are shown, in figures 6 (US English) and 7 (French).

```
$ American English message catalog for example 1
$set 1
1 Cannot divide %lg by %lg: divisor is too small.
2 Cannot take arcsin of %lg: value out of range.
3 Cannot take square root of %lg: argument is negative.
4 Cannot calculate F(%lg, %lg, %lg)
```

*Figure #6:* `example/en_US.iso88591.msg`

```
$ French message catalog for example 1
$set 1
1 Division impossible de %lg par %lg: diviseur trop petit.
2 arcsin de %lg impossible: valeur hors des limites.
3 Racine carrée de %lg impossible: argument négatif.
4 Calcul impossible de F(%lg, %lg, %lg).
```

*Figure #7:* `example/fr_FR.iso88591.msg`

We have adopted the recommended message catalog naming scheme, which suggests names of the form "*language_country*.*charset*.msg" where *language* is the two–letter language identifier specified in ISO 639, *country* is the two–letter country identifier specified in ISO 3316, and *charset* is, in the SL Control System, always `iso88591`. A "`.msg`" file is compiled into a "`.cat`" file with the `gencat(1)` command; the "`.cat`" file is in a form accessable by `catopen(3)`.

## 3.2 Exceptions

Figure 5 illustrates a style problem that often comes up when error checking is rigorously applied: there are many places where an error can occur, all or most of which require the same treatment. In the example above, we used `goto` to avoid clutter, but many people shun such usage. A more common approach would have been to use the code fragment

```
{
    return error_stack(e, ERROR_EXAMPLE_F,
        "Cannot calculate F(%lg, %lg, %lg).", a, b, v);
}
```

as the body of each `if` statement, but this has the drawback of repeating the same code in many places.

Therefore, we provide an exception handler with the SL Error System. The exception handler can be used independently of the Error System, and the use of the Error System does not require the developer to use the exception handler (though the Error System uses it internally). However, the two systems complement each other nicely, and the error numbers already introduced may be used as the exceptions.

This exception handler is based on the one proposed for the C++ language. Subsequent to its implementation the C++ proposal changed somewhat (e.g. *raise* was changed to *throw*), but this one has not been changed.

In short, an *exception number* is any 31–bit number[2]. At any point, a call to the `raise` function, with an exception number as the parameter, will transfer control of execution to a related `catch` or `catchall` label.

The exception handler uses six statements. Five are C preprocessor macros, and one is a function call. They are defined in the header file `exception.h`, available in `$SL_INCLUDE`. The six statements are:

! `try`
This begins a block of protected code.

! `endtry`
This ends a block of protected code.

! `catch(`*n*`):`
This marks the beginning of a section of code devoted to handling exception number *n*.

! `catchall`:
This marks the beginning of a section of code devoted to handling any exception in the current `try`/`endtry` block not handled by any other `catch` statement.

! `raise(`*n*`)`
This raises an exception, i.e., it causes the flow of code to jump to the `catch` or `catchall` statement of the innermost `try`/`endtry` block that is prepared to handle this exception.

! `reraise`
This statement, valid only after a `catch` or `catchall`, re–raises the current exception to the next `try`/`endtry` block prepared to handle it.

The basic mechanism is illustrated in figure 8.

```
try      /* Begin protected block */
{
    /* code here */
    if( error situation ) raise(1);

    /* more code */
    if( error situation ) raise(2);

    /* more code */
    if( error situation ) raise(3);

    /* more code */
    if( error situation ) raise(2);
}        /* End protected block */
catch(2):
    /* handle situation 2 */
    /* fall through */
catch(1):
    /* handle situation 1 */
    break;    /* continue after the endtry */
catchall:
    /* handle all other excpteions */
endtry;
```

*Figure #8*

2. 32–bit integers are used, but for implementation reasons the high bit must be zero. This means that a modification will have to be made when we find ourselves with thirty–two thousand products in SLAPS.

The code between the `try` and the first `catch` is said to be *protected by an exception handler.* Within this block of code, any `raise` will cause the normal flow of execution to be interrupted, with execution resuming at a `catch` or `catchall` statement.

In the case of the first error situation, `raise(1)` is called. This causes the flow of execution to jump to the `catch(1):` label below. There, there is the code to handle this situation, and the `break` causes the flow to be resumed after the `endtry`. Using the line numbers included in the figure, the flow of control follows the following lines: *1, 2, 3, 4, 18, 19, 20, 24.*

In the case of the second error situation, `raise(2)` is called. This causes the flow to jump to the `catch(2):` label below. The code after `catch(2):` does not end with a `break`, so the flow continues into the next `catch` area (like the C `case` label). The lines executed are: *1, 2, 3, 4, 5, 6, 7, 15, 16, 17, 18, 19, 20, 24.*

In the third error situation, `raise(3)` is called. There is no `catch(3):` below, but there is a `catchall:`. Therefore, execution continues there: *1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 21, 22, 23, 24.*

The fourth error situation merely shows that the same exception can be raised in multiple places. As in the second example, execution jumps from the `raise(2)` call to line 15.

The "normal" flow of execution, the one without any errors, is this: *1, 2, ... 13, 14, 23, 24.*

`try`/`endtry` blocks may be nested. If the same exception is caught in more than one of these nested blocks, the *innermost* `catch` is used. If that block executes the `reraise` statement, that same exception will be re–raised to the next level.

Any exceptions not caught with `catch` or `catchall` are automatically re–raised to the next block. If there is no next block, this uncaught exception will cause the program to abort. Every `try` must have a corresponding `endtry`; the code will not compile without it. `catch` and `catchall` are optional in any given block; however, every exception should be caught at some level.

There is one restriction when using the exception handler: do not `return`, `continue`, or `goto` out of the code between the `try` statement and the first `catch` statement. These statements may be used to transfer control *within* this block, but not *out* of it. These statements may be freely used in the `catch` and `catchall` blocks. Additionally, within the `catch` and `catchall` blocks, the C statement `continue` will cause the `try` block to be re–executed from the beginning. In the `try` block, a `break` will continue execution after the `endtry`.

The greatest value of the exception handler is that `raise`/`catch` can be used to transfer control out of a subroutine.

A more advanced example of the exception handler is provided in figure 9.

This example contains two nested `try`/`endtry` blocks, one from lines *3* through *29*, and the other from lines *7* through *12*.

The innermost one merely protects the "`map_file`" routine. If this routine raises the `EXC_OUT_OF_ROOM` exception (presumably `#define`'d in a header file somewhere), it is caught by the `catch` at line 9, and the "`increase_quota`" routine is called. The `continue` statement then directs that flow return to line *7*, and "`map_file`" is tried again. This can loop indefinitely. If `map_file` raises any other exception, then since the inner `try`/`endtry` does not catch it, the exception is passed to the outer `try`/`endtry` block. If `increase_quota` raises any exception, including `EXC_OUT_OF_ROOM`, then that also will be passed to the outer block.

If any of the routines `bar`, `baz`, or `quux` raise exceptions, control will jump out of the routine to one of the `catch` statements below. The code following the `catch` statements on lines *18*

```
1  void foo(char *file)
2  {
3      try
4      {
5          char *mapping;
6
7          try
8              mapping = map_file(file);
9          catch(EXC_OUT_OF_ROOM):
10             increase_quota();
11             continue;
12         endtry;
13
14         bar();
15         baz(mapping);
16         quux();
17     }
18     catch(EXC_NO_FILE):
19         fprintf(stderr, ”File %s not found.\n”, file);
20         reraise;
21     catch(EXC_OUT_OF_ROOM):
22         fprintf(stderr, ”File %s is too big.\n”, file);
23         reraise;
24     catch(EXC_BAR_FAILED):
25         fprintf(stderr, ”bar() failed.\n”);  /* fall through */
26     catchall:
27         fprintf(stderr, ”Foo failed on file %s.\n”, file);
28         reraise;
29     catch(EXC_FILE_CONVERTED):
30         printf(”File %s has already been converted!”, file);
31         break;
32     endtry;
33 }
```

*Figure #9*

and *21* will print out an error message, and re–raise the same exception to whatever called `foo`. The `catch` at line *24* will also catch a specific exception, print a message, and fall through to the default case. The default `catchall` prints a default error message and also re–raises the exception. The final `catch` catches an exception which is not to be considered a failure of `foo`, so it prints an informational message and, by using `break`, jumps out of the `try`/`endtry` block.

## 3.3 Errors and exceptions

This exception mechanism complements the SL Error System nicely. The common error–handling code, which frequently will consist of stacking another error message onto an error stack, can be placed in a `catchall` block, where it will not clutter up the main code. The error number is a convenient exception number.

Because an exceptional return from a subroutine does not allow any return value (aside from the exception number itself), the Error System must provide a way of using an external error stack. Therefore, each error library call of the form `error_`*something* has an equivalent named `Error_`*something*. The equivalent calls are identical, except that the first parameter, the error stack, is omitted. There are also some added routines to get and put the external error stack, to allow the two sets of calls to be mixed. The header file `error_handler.h` is replaced by `Error_handler.h`.

It should be noted that on *multithreaded* platforms, both the exception system and the error system are *thread–safe*.

Figure 10 shows the first example, re–implemented to use the exception mechanism. The "wrapper" functions, which safely perform the sub–steps, are somewhat more complex, but the higher–level function `F` is very simple. Full error checking is being done, but it does not get in the way of the "normal" code.

## 3.4 Standard wrapper functions

One of the major sources of possible errors is the set of standard system and library calls. The SL Error System comes with a set of *wrappers* for these calls. The libraries of wrapper calls are complete for the Apollo and HP–UX system calls, and as of this writing there are incomplete libraries for the Apollo and HP–UX library calls, and the LynxOS system and library calls. Due to time constraints, the incomplete libraries are being completed only as needed.

Each wrapper call has the same name as its underlying system or library call, except that the first letter is capitalized (e.g., "`Fopen`" or "`Malloc`."). All parameters are the same as the underlying call. If the underlying call returns a value only for the purpose of error checking, the wrapper "returns" void; otherwise, the return value also matches the underlying call. All errors start an error stack and raise an exception. For system calls which set `errno`, the error stack created is two messages deep: the first corresponds to the `errno` value, the second is a report that the system call failed. Other calls create a stack with one message, reporting the failure of that call. The prototypes of these calls, and the definitions of the error numbers, are provided in the include files `Libcalls.h` and `Syscalls.h`, available in `$SL_INCLUDE`. The libraries, as well as the exception library, are included in the `$SL_ERRORLIB` set of libraries.

The use of these libraries often makes the writing of code with rigorous error checking as simple as ignoring the errors completely. For example, figure 11 shows a simple program that connects to a named TCP/IP service on a named host, and sends to standard output all data provided by the service. Full error checking is done at every call.

## 3.5 Analyzing errors

### 3.5.1 Converting error stacks to text

There are situations where one may wish to convert an error stack into a plain–text human–readable version. For instance, a server program may have an "ASCII mode" so that it may be accessed with `telnet`. In normal situations, a server should pass back an error stack to the client using the import/export calls described below. However, a person telnetting to the server will not be able to understand the "raw" error data. Therefore, in this case, the server should convert any error to text before it is sent.

There is a call to do this conversion, called `error_text`. It takes two parameters, an error stack, and a pointer to a `char **`. The value pointed to by the second parameter is set to a null–terminated list of pointers to standard C strings. The user should free the allocated memory (the strings and the list of pointers) after use. This call can return an error stack in the case of memory exhaustion.

To illustrate this call, figure 12 shows a simple routine which prints an error stack on `stderr`.

```
double divide
(
    double dividend,
    double divisor
)
{
    if( fabs(divisor) < EPSILON )
    {
        Error_stack(ERROR_EXAMPLE_DIV_BY_ZERO,
            "Cannot divide %lg by %lg: divisor is too small.", dividend, divisor);
        raise(ERROR_EXAMPLE_DIV_BY_ZERO);
    }

    return dividend/divisor;
}

double arcsin
(
    double x
)
{
    if( x < -1.0 || x > 1.0 )
    {
        Error_stack(ERROR_EXAMPLE_ASIN_RANGE,
            "Cannot take arcsin of %lg: value out of range.", x);
        raise(ERROR_EXAMPLE_ASIN_RANGE);
    }

    return asin(x);
}

double root
(
    double s
)
{
    if( s < 0.0 )
    {
        Error_stack(ERROR_EXAMPLE_SQRT_NEGATIVE,
            "Cannot take square root of %lg: argument is negative.", s);
        raise(ERROR_EXAMPLE_SQRT_NEGATIVE);
    }

    return sqrt(s);
}

double F
(
    double a,
    double b,
    double v
)
{
    double r;

    try
        r = divide( arcsin( divide(a, b) ), root( 1 - v*v ) );
    catchall:
        Error_stack(ERROR_EXAMPLE_F, "Cannot calculate F(%lg, %lg, %lg).", a, b, v);
        reraise;
    endtry;
    return r;
}
```

*Figure #10*

```
#include ”Error_handler.h”
#include ”Syscalls.h”
#include ”Libcalls.h”
#include <sys/types.h>
#include <netinet/in.h>

void main(int argc, char *argv[])
{
    char *host = (argc > 1) ? argv[1] : ”localhost”;
    char *serv = (argc > 2) ? argv[2] : ”daytime”;
    struct hostent *hp;
    struct servent *sp;
    struct sockaddr_in sa;
    int s, l;
    char buffer[ 1024 ];

    try
    {
        hp = Gethostbyname(host);
        bcopy((char *)hp->h_addr, (char *)&sa.sin_addr, hp->h_length);
        sa.sin_family = hp->h_addrtype;
        sp = Getservbyname(serv, ”tcp”);
        sa.sin_port = sp->s_port;
        s = Socket(hp->h_addrtype, SOCK_STREAM, 0);
        Connect(s, (struct sockaddr *)&sa, sizeof(sa));

        while( l = Read(s, buffer, sizeof(buffer)) )
            Write(1, buffer, l);
    }
    catchall:
        Error_output();
    endtry;
}
```

*Figure #11*

```
error_t pslerror(error_t e)
{
    if( e )
    {
        char **text, **x;

        error_t e2 = error_text(e, &text);
        if( e2 != error_ok )
        {
            return error_stack(e2, ERROR_EXAMPLE_PSLERROR,
                ”Cannot print an error stack at 0x%08x to stderr.”, e);
        }

        for( x = text; *x; x++ )
        {
            fprintf(stderr, %s\n”, *x);
            free(*x);
        }

        free(text);
    }

    return error_ok;
}
```

*Figure #12*

The text produced contains one line per error message. The text on this line begins with the timestamp of the error, a standard message of the form, "`Error number` *number* `at line` *n* `of file` *filename*`:`," and the error message itself. This standard message, and each error message, will if possible be in the language identifed by the `LANG` environment variable. Each error message will have all of its parameters properly converted and inserted.

### 3.5.2 Error numbers in stacks

In some situations, a program can recover from some errors. Therefore, when an operation fails, the error stack must be examined to see if the situation is recoverable. This is the primary function of the error numbers.

To find out the error number of the bottommost error message (the first one stacked) on the stack, use the `error_bottom` macro. It takes one parameter, the error stack; and returns the number. To find out the error number of the topmost (most recent) number, use `error_top.` To find the number of any particular error message, use `error_nth`. `error_nth` takes two parameters, the error stack and the index number. The index number is zero for the bottommost message, and successive integers for higher messages. The number of messages in a stack may be determined with `error_depth`. Figure 13 illustrates these calls with an example routine.

```
void error_numbers(error_t e)
{
    if( e )
    {
        error_number_t b = error_bottom(e);
        error_number_t t = error_top(e);
        int n = error_depth(e), i;

        printf("An error stack has been received.\n");
        printf("\tAt the root of the problem is error 0x%08x.\n", b);

        if( n > 1 )
            printf("\tThe end effect is error 0x%08x.\n", t);

        if( n > 2 )
        {
            printf("\tAnd in-between there are the errors:\n");
            for( i = 1; i < (n-1); i++ )
                printf("\t\t0x%08x\n", error_nth(e, i));
        }
    }
}
```

*Figure #13*

## 3.6 Interprocess error communication

Processes which cooperate to perform a task will need the ability to send error stacks between themselves. For instance, if a client makes a request of a server which results in an error, the server will need to be able to communicate this error stack back to the client.

### 3.6.1 RPC

The SL RPC system has been modified to pass SL Error System error objects. In the `.nc` file defining the RPC relationship, define the `error_t` type with the following line:

```
error_t: type = private;
```

This type may then be used like any other RPC type. In particular, server functions may be defined as returning `error_t`.

The SL RPC system indicates a call failure with a special return value, which may be specified in the `.nc` file. Since the Error System's *error_ok* value is zero, the `.nc` file should not specify that the error return value be zero. In the future, the RPC system may allow the option of indicating RPC errors with error objects and/or exceptions.

### 3.6.2 File descriptors

Programs which communicate over sockets or pipes need a way of sending and receiving error stacks over file descriptors. The `error_export` and `error_import` routines do this.

The `error_export` routine sends an error stack. It takes two parameters, the error stack and the file descriptor, and returns an error stack. `error_import` receives an error stack. It takes one parameter, the file descriptor, and returns an error stack. Please note that `error_import` can return a non–*error_ok* error stack for two reasons: it failed, and generated its own error stack, or it succeeded, and is returning the error stack sent. These can be distinguished by looking at the error bases of the message(s) in the stack: if they are all of the base `ERROR_ERROR_BASE`, the `error_import` failed.

`error_import` does not do any explicit verification that it is receiving a valid error. The two processes must have already established that an error is to be sent. For example, a server process should *not* be written to return an error stack upon error, and a block of data otherwise: in this case, the client will not know what to read. Instead, the server could be written to always return an error stack, and if the error sent is *error_ok*, then the data follows. This way the client will know to always call `error_import`, and if that returns *error_ok*, it reads the result.

These calls are illustrated in figures 14 and 15. Figure 14 shows a hypothetical server routine, which receives a request, processes it, sends back the error stack that resulted from the processing, and if the processing returned *error_ok*, the answer is sent back as well. Figure 15 shows the matching client, which sends a request, receives the error stack, and receives the answer. If any step of the client fails, it returns the resulting error stack.

### 3.6.3 `FILE` pointers

There also exist versions of the export and import routines which use the buffered "`FILE *`" type defined in `<stdio.h>`. They are called `error_fexport` and `error_fimport`.

```
error_t serve(int socket)
{
    error_t e, e2;
    struct request r;
    struct answer a;

    e = receive_request(socket, &r);
    if( e != error_ok ) return e;

    e = process_request(r, &a);
    e2 = error_export(e, socket);

    if( e2 != error_ok )
    {
        error_free(e);  /* We have worse problems than a bad request */
        return e2;
    }

    if( e == error_ok )
        e = send_answer(a, socket);

    return e;
}
```

*Figure #14*

```
error_t request(int socket, struct request r, struct answer *ap)
{
    error_t e;

    e = send_request(r, socket);
    if( e != error_ok ) return e;

    e = error_import(socket);
    if( e != error_ok ) return e;

    e = receive_answer(socket, ap);
    return e;
}
```

*Figure #15*

Their declarations are identical with `error_export` and `error_import`, except that the integer file descriptors are replaced with “`FILE *`” file pointers.

### 3.6.4 SL–EQUIP package

At the time of writing, modifications to the SL–EQUIP package are being considered which will make it possible for *equipment servers* to return error stacks to application programs. It is possible that future version of SL–EQUIP may use the SL Error System to report errors.

## 3.7 Reporting errors to the error server

At the “top” of the software chain, there will be a program which either was directly invoked by a person, or invoked automatically via some system mechanism (e.g., `cron`, `inetd`, `init`, or one of several SL/CO server starting methods). As was discussed in section 2, this top–level program must send any errors to the appropriate error server.

This is done with one call, `error_output`. (Like all `error_`*something* routines, there is a corresponding call for use with the exception system. In this case, it is `Error_output`.) By definition, `error_output` *takes full responsibility for reporting an error.*

The original design tried the following:

- ! If there is an error server running on the same computer, the error is delivered to it. If not,
- ! The general error server is contacted, and the error is delivered to it. If this fails (for instance, in the case of a network partition),
- ! The text is sent to `stderr`. This isn't ideal, but there's not much else to do in a partition situation.

The current implementation uses the SPS Error Server. This is one of the areas being actively developed. This document should be changed when the SL Error Server is put into operation.

## 3.8 Using the SL Error Server

Currently, the SL Error System ties into the SPS Error Server. See SPS/AOP/OS/Note 46 for instruction on using the SPS Error Server. When the final form of the SL Error Server is released, this section will be updated.

# 4 The SL Error System Reference Manual

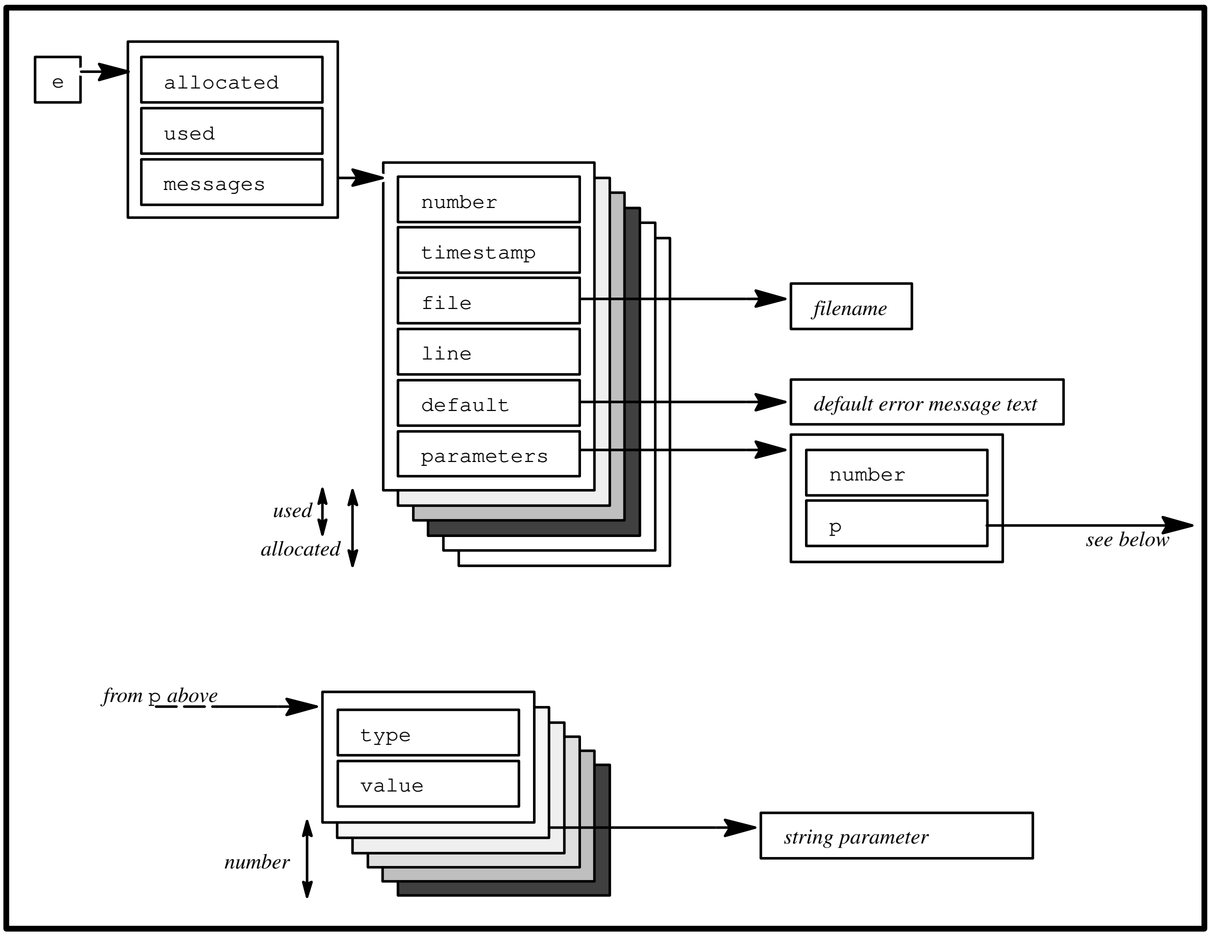
## 4.1 Data types

### 4.1.1 error_number_t

This is the type of an *error number*. This type is defined in `error_handler.h`. It is an unsigned 32–bit integer. The most significant 16 bits identify a particular software package. The least significant 16 bits identify a particular error within the specified package. The error number consisting of the high 16 bits of a package, with the low 16 bits set to zero, identifies the package, and is referred to as an *error base*. The error number identical to zero indicates a “no error” situation.

### 4.1.2 error_t

This is the type of an *error stack*, also called an *error object*. This type is defined in `error_handler.h`. Its internal structure is intended to be opaque; all access should be through SL Error Library routines. However, as an aid to understanding, this is the structure of the error object at the time of writing:



Given a variable *e* of type `error_t`: *e* is a pointer to a structure with three elements. The third, `messages`, points to an array of structures. The first element is the allocated size of the array, the second is the number of elements used. Each element corresponds to one error message. Each structure in the array has six elements: the error number, the timestamp of the error, a pointer to a null–terminated string containing the filename containing the `error_stack` (or `Error_stack`) call that stacked that message, the line number of that call, a pointer to a null–terminated string containing the default message, and a pointer to the message parameters. This last pointer points to a structure which merely contains the number of parameters, and a pointer to an array of structures. There is one element per parameter. Each element contains a type flag which identifies the parameter type, and a union containing the actual parameter value. If the parameter is a string, the value is a pointer to a standard null–terminated string.[3]

### 4.1.3 `exception_t`

This is the type of an *exception*, and is defined in `exception.h`. It is a signed 32–bit integer with the added restriction that it must be positive (i.e., the high bit must be zero, and the value cannot be zero). Error numbers are ideal exceptions, as long as no product is assigned an error base over `0x7FFF0000`.

## 4.2 Error library

Some of the "routines" listed below are actually C preprocessor macros. These are clearly marked.

When a routine is said to return an error stack, a list of possible errors is given. For each item in this list, the mnemonic name of the error number of the topmost error message is given. The underlying error messages, if any, will depend on the circumstances of the problem and are not given here.

It should be pointed out that except for machine word limits (e.g., the requirement that the error number be a 32–bit integer), there are no arbitrary hard limits in the SL Error System. In particular, the length of the error message (either converted or unconverted) may be of any length for which the computer has enough memory and disk space to contain. The number of error messages in an error stack, and the number of parameters in an error message, must both be stored in 32–bit integers, but a process would run out of address space long before these limits are reached.

3. If anybody ever re–implements a system like this, I would suggest an addition: there should be a multi-lingual character string option. Perhaps the conversion specifier could be "`%S`" (capital S). There would actually be three parameters that correspond to that one specifier: two numbers, which would identify the message catalog set number and message number, and a default string. I haven't added this, since the SL Error System is no longer my project.

# 4.2.1 `error_bottom`

## 4.2.1.1 Functional description

`error_bottom` returns the *error number* of the *bottommost error message* of an *error stack.*

## 4.2.1.2 Cautions and warnings

! `error_bottom` *is a C preprocessor macro.*

## 4.2.1.3 Formal description

```
error_number_t
error_bottom
(
    error_t e
);
```

### 4.2.1.3.1 Return value

This macro returns an *error number,* of type `error_number_t`.

### 4.2.1.3.2 Parameters

`e` This parameter is an *error stack*, of type `error_t`. It specifies the *error stack* of which the *error number* of the *bottommost error message* is returned. This parameter is required, and there is no default.

### 4.2.1.3.3 Special behavior

This macro returns a zero when its parameter is `error_ok`. The results are undefined if it is not passed a valid *error stack.*

## 4.2.1.4 Examples

```
{
    error_t e;
    error_number_t n;

    e = error_stack(NULL, 0x12340001, "Message one.");
    e = error_stack(e,    0x12340002, "Message two.");
    e = error_stack(e,    0x12340003, "Message three.");

    n = error_bottom(e);

    assert( n == 0x12340001 );

    error_free(e);
}
```

## 4.2.1.5 Errors

`error_bottom` does not return any errors.

## 4.2.1.6 References to related operations

See also,

- `error_nth`
- `error_top`
- `Error_bottom`

## 4.2.2 `error_depth`

### 4.2.2.1 Functional description

`error_depth` returns the number of *error messages* in an *error stack.*

### 4.2.2.2 Cautions and warnings

! `error_depth` *is a C preprocessor macro.*

### 4.2.2.3 Formal description

```
int
error_depth
(
    error_t e
);
```

#### 4.2.2.3.1 Return value

This macro returns an `int`.

#### 4.2.2.3.2 Parameters

`e` This parameter is an *error stack,* of type `error_t`. It specifies the *error stack* of which the number of *error messages* is returned. This parameter is required, and there is no default.

#### 4.2.2.3.3 Special behavior

This macro returns a zero when its parameter is `error_ok`. The results are undefined if it is not passed a valid *error stack.*

### 4.2.2.4 Examples

```
{
    error_t e;
    int depth

    e = error_stack(NULL, 0x12340001, "Message one.");
    e = error_stack(e,    0x12340002, "Message two.");
    e = error_stack(e,    0x12340003, "Message three.");

    depth = error_depth(e);

    assert( n == 3 );

    error_free(e);
}
```

### 4.2.2.5 Errors

`error_depth` does not return any errors.

### 4.2.2.6 References to related operation

See also,

- `error_nth`
- `Error_depth`

### 4.2.3 `error_export`

#### 4.2.3.1 Functional description

`error_export` sends an *error stack* over a file descriptor.

#### 4.2.3.2 Cautions and warnings

! `error_export` no longer releases the storage associated with the *error stack* passed to it after it has been successfully exported.

#### 4.2.3.3 Formal description

```
error_t
error_export
(
    error_t e,
    int fd
);
```

##### 4.2.3.3.1 Return value

This function returns an *error stack*. If it fails, the *error number* of the topmost *error message* will be one of the values listed in 4.2.3.5 Upon successful completion, this *error stack* has the value `error_ok`.

##### 4.2.3.3.2 Parameters

`e` This parameter is an *error stack*, of type `error_t`. It specifies the *error stack* which is to be exported.

`fd` This parameter is a *file descriptor,* of type `int`. It specifies the socket, file, pipe, or other filesystem object to which the *error stack* is to be exported.

##### 4.2.3.3.3 Special behavior

The special *error stack* `error_ok` is a perfectly valid parameter, and will be exported properly.

#### 4.2.3.4 Examples

```
{
    int p[2];
    if( pipe(p) == -1 )
        return error_stack(NULL, ERROR_EXAMPLE_PIPE, "pipe() failed.");
    switch( fork() )
    {
        case -1:  /* error */
            return error_stack(NULL, ERROR_EXAMPLE_FORK, "fork() failed.");
        case :0:  /* child */
        {
            error_t e;
            e = error_stack(NULL, 0x12340011, "Message one.");
            e = error_stack(e,    0x12340012, "Message two.");
            e = error_export(e, p[1]);
            if( e != error_ok )
                error_output(e);
            exit(0);
        }
        default:  /* parent */
        {
            error_t e = error_import(e, p[0]);
            error_output(e);
            return error_ok;
         }
    }
}
```

#### 4.2.3.5 Errors

`error_export` can fail for two reasons:

- `ERROR_ERROR_NO_MEMORY`: The routine cannot temporarily allocate enough memory to marshall the data.
- `ERROR_ERROR_WRITE`: An underlying call to `write(2)` failed.

In the case of an error, an *error stack* will be returned. The *error number* of the topmost *error_message* will be one of the two constants listed above.

#### 4.2.3.6 References to related operations

See also,

- `error_fexport`
- `error_import`
- `error_output`
- `Error_export`

### 4.2.4 error_fexport

#### 4.2.4.1 Functional description

error_fexport writes an *error stack* to a file identified by a “FILE *” file pointer.

#### 4.2.4.2 Cautions and warnings

! error_fexport no longer releases the storage associated with the *error stack* passed to it, after it has been successfully exported.

#### 4.2.4.3 Formal description

```
error_t
error_fexport
(
    error_t e,
    FILE *fp
);
```

##### 4.2.4.3.1 Return value

This function returns an *error stack*. If it fails, the *error number* of the topmost *error message* will be one of the values listed in 4.2.4.5 Upon successful completion, this *error stack* has the value error_ok.

##### 4.2.4.3.2 Parameters

e This parameter is an *error stack*, of type error_t. It specifies the *error stack* which is to be exported.

fp This parameter is a *file pointer,* of type FILE *. It specifies the filesystem object to which the *error stack* is to be exported.

##### 4.2.4.3.3 Special behavior

The special *error stack* error_ok is a perfectly valid parameter, and will be exported properly.

#### 4.2.4.4 Examples

```
{
    error_t e;
    FILE *fp;

    e = error_stack(NULL, 0x12340021, "First message.");
    e = error_stack(e,    0x12340022, "Second message.");

    fp = fopen("logfile.err", "w");
    if( fp != NULL )
    {
        error_t e2 = error_fexport(e2, fp);
        if( e2 != error_ok )
            error_free(e);
        return e2;
    }

    error_free(e);
    return error_stack(NULL, ERROR_EXAMPLE_FOPEN, "Cannot fopen %s.", "logfile.err");
}
```

#### 4.2.4.5 Errors

`error_fexport` can fail for two reasons:

- `ERROR_ERROR_NO_MEMORY`: The routine cannot temporarily allocate enough memory to marshall the data.
- `ERROR_ERROR_FWRITE`: An underlying call to `fwrite(3)` failed.

In the case of an error, an *error stack* will be returned. The *error number* of the topmost *error_message* will be one of the two constants listed above.

#### 4.2.4.6 References to related operations

- `error_export`
- `error_fimport`
- `Error_fexport`

### 4.2.5 error_fimport

#### 4.2.5.1 Functional description

error_fimport reads an *error stack* from a file identified by a “FILE *” file pointer.

#### 4.2.5.2 Cautions and warnings

! error_fimport may return a non–error_ok *error stack* either when it encounters an error and cannot function, or when it successfully reads a non–error_ok stack. These two cases may usually be distinguished by examining the *error number* of the topmost *error message* of the returned stack.

#### 4.2.5.3 Formal description

```
error_t
error_fimport
(
    FILE *fp
);
```

##### 4.2.5.3.1 Return value

This function returns an *error stack*. If it fails, the *error number* of the topmost *error message* will be one of the values listed in 4.2.5.5. If it succeeds, the *error stack* returned will be the one read. Only if this routine successfully reads in an error_ok error stack will it return error_ok.

##### 4.2.5.3.2 Parameters

fp This parameter is a *file pointer,* of type FILE *. It specifies the filesystem object from which the *error stack* is to be imported.

##### 4.2.5.3.3 Special behavior

None.

#### 4.2.5.4 Examples

```
{
    FILE *fp = fopen("logfile.err", "r");
    if( fp == NULL )
        return error_stack(NULL, ERROR_EXAMPLE, FOPEN, "Cannot fopen logfile.err.");

    while( 1 )
    {
        error_t e = error_fimport(fp);

        if( (error_top(e) & 0xFFFF0000) == ERROR_ERROR_BASE )
        {
            fclose(fp);
            return e;
        }

        feed_histograms(e);
        error_free(e);
    }
}
```

#### 4.2.5.5 Errors

`error_fimport` can fail for two reasons:

- `ERROR_ERROR_NO_MEMORY`: The routine cannot allocate enough memory to create the *error stack.*
- `ERROR_ERROR_FREAD`: An underlying call to `fread(3)` failed.

In the case of an error, an *error stack* will be returned. The *error number* of the topmost *error_message* will be one of the two constants listed above.

#### 4.2.5.6 References to related operations

- `error_fexport`
- `error_import`
- `Error_fimport`

## 4.2.6 error_free

### 4.2.6.1 Functional description

error_free frees all allocated space associated with an *error stack*.

### 4.2.6.2 Cautions and warnings

! After it has been freed, any references to or uses of an *error stack* will result in undefined behavior.

### 4.2.6.3 Formal description

```
void
error_free
(
    error_t e
);
```

#### 4.2.6.3.1 Return value

This function does not return any value.

#### 4.2.6.3.2 Parameters

e This parameter is an *error stack*, of type error_t. It specifies the *error stack* which is to be freed.

#### 4.2.6.3.3 Special behavior

Freeing an *error stack* of value error_ok does not accomplish anything, but it is perfectly safe to do it.

### 4.2.6.4 Examples

```
{
    error_t e = subroutine();

    if( e != error_ok )
    {
        if( error_top(e) == ERROR_EXAMPLE_WARNING )
            error_free(e);
        else
            return error_stack(e, ERROR_EXAMPLE_FAILED,
                "subroutine failed with more than a warning.");
    }

    /* ... */
}
```

### 4.2.6.5 Errors

This function does not have any errors.

### 4.2.6.6 References to related operations

- error_stack
- Error_free
- Error_stack

### 4.2.7 `error_import`

#### 4.2.7.1 Functional description

`error_import` receives an *error stack* over a socket, pipe, or other file descriptor.

#### 4.2.7.2 Cautions and warnings

! `error_import` may return a non–`error_ok` *error stack* either when it encounters an error and cannot function, or when it successfully reads a non–`error_ok` stack. These two cases may usually be distinguished by examining the *error number* of the topmost *error message* of the returned stack.

#### 4.2.7.3 Formal description

```
error_t
error_import
(
    int fd
);
```

##### 4.2.7.3.1 Return value

This function returns an *error stack*. If it fails, the *error number* of the topmost *error message* will be one of the values listed in 4.2.7.5. If it succeeds, the *error stack* returned will be the one read. Only if this routine successfully reads in an `error_ok` error stack will it return `error_ok`.

##### 4.2.7.3.2 Parameters

`fd` This parameter is a *file descriptor*, of type `int`. It specifies the socket, pipe, or other file descriptor over which the *error stack* is to be read.

##### 4.2.7.3.3 Special behavior

The function may return a non–`error_ok` *error stack* even when it works properly.

### 4.2.7.4 Examples

```
{
    int p[2];
    if( pipe(p) == -1 )
        return error_stack(NULL, ERROR_EXAMPLE_PIPE, "pipe() failed.");
    switch( fork() )
    {
        case -1:  /* error */
            return error_stack(NULL, ERROR_EXAMPLE_FORK, "fork() failed.");
        case :0:  /* child */
        {
            error_t e;
            e = error_stack(NULL, 0x12340011, "Message one.");
            e = error_stack(e,    0x12340012, "Message two.");
            e = error_export(e, p[1]);
            if( e != error_ok )
                error_output(e);
            exit(0);
        }
        default:  /* parent */
        {
            error_t e = error_import(e, p[0]);
            error_output(e);
            return error_ok;
         }
    }
}
```

### 4.2.7.5 Errors

`error_import` can fail for two reasons:

- `ERROR_ERROR_NO_MEMORY`: The routine cannot allocate enough memory to create the *error stack.*
- `ERROR_ERROR_READ`: An underlying call to `read(3)` failed.

In the case of an error, an *error stack* will be returned. The *error number* of the topmost *error_message* will be one of the two constants listed above.

### 4.2.7.6 References to related operations

See also,

- `error_export`
- `error_fimport`
- `Error_import`

## 4.2.8 `error_nth`

### 4.2.8.1 Functional description

`error_nth` returns the *error number* of one specified *error message* of an *error stack.*

### 4.2.8.2 Cautions and warnings

! `error_depth` *is a C preprocessor macro.*

### 4.2.8.3 Formal description

```
error_number_t
error_nth
(
    error_t e,
    int n
);
```

#### 4.2.8.3.1 Return value

This macro returns an *error number,* of type `error_number_t`.

#### 4.2.8.3.2 Parameters

`e` This parameter is an *error stack*, of type `error_t`. It specifies the *error stack* of which the *error number* of the *bottommost error message* is returned.

`n` This parameter is an index, of type `int`. It specifies the *error message* of which the *error number* is to be returned. Zero corresponds to the bottommost message, `(error_depth(e)-1)` corresponds to the topmost.

#### 4.2.8.3.3 Special behavior

This macro returns a zero when its first parameter is `error_ok`. The results are undefined if it is not passed a valid *error stack,* or if the index is out of the range `[0, error_depth(e))`.

### 4.2.8.4 Examples

```
{
    error_t e;
    error_number_t n;

    e = error_stack(NULL, 0x12340001, "Message one.");
    e = error_stack(e,    0x12340002, "Message two.");
    e = error_stack(e,    0x12340003, "Message three.");

    n = error_nth(e, 1);

    assert( n == 0x12340002 );

    error_free(e);
}
```

### 4.2.8.5 Errors

`error_nth` does not return any errors.

#### 4.2.8.6 References to related operations

See also,

- `error_bottom`
- `error_depth`
- `error_top`
- `Error_nth`

## 4.2.9 error_output

### 4.2.9.1 Functional description

error_output communicates an *error stack* to an *error server,* if one is running on the same host. Usually, computers with displays that run programs interactively will run an *error server.*

If there is not an *error server* running locally, the behavior is not yet defined. The original design of *the SL Error System* called for this routine to transmit the *error stack* to a *general error server* in the control room, which would display the errors of important background servers on an *Alarm System*–like display. However, this is one of the subjects which is being reconsidered by the new *Error System* developers, and it will have to be clarified in a future revision of this document.

If all else fails, and the *error stack* cannot be sent to an *error server* for appropriate handling, the text of the *error stack* is sent to stderr. By definition, this routine takes full responsibility for reporting an error.

### 4.2.9.2 Cautions and warnings

! The original definition of error_output freed the *error stack's* storage after successfully sending it. This was subsequently changed, so now error_output does *not* free the storage space. If you do not have a further use for an *erorr stack* (such as logging it), remember to call error_free.

### 4.2.9.3 Formal description

```
void
error_output
(
    error_t e
);
```

#### 4.2.9.3.1 Return value

This function does not return anything.

#### 4.2.9.3.2 Parameters

e This parameter is an *error stack*, of type error_t. It specifies the error that is to be reported by the *error server.*

#### 4.2.9.3.3 Special behavior

At the time of writing, the *SL Error Server* is not yet operational. This routine currently uses the *SPS Error Server* instead. See SPS/AOP/OS/Note 46 for more details about the *SPS Error System.*

### 4.2.9.4 Examples

```
{
    error_t e;

    e = error_stack(NULL, 0x12340030, "Bottom message.");
    e = error_stack(e,    0x12340031, "Middle message.");
    e = error_stack(e,    0x12340032, "Top message.");

    error_output(e);
    error_free(e);
}
```

#### 4.2.9.5 Errors

This routine does not produce any errors.

#### 4.2.9.6 References to related operations

See also,

- `error_text`
- `Error_output`

# 4.2.10 `error_package_name`

## 4.2.10.1 Functional description

`error_package_name` returns the pathname of a SLAPS product's *message catalog* for the language specified by the current process's environment. This is useful when a product uses the *message catalog* not only for internationalized error messages, but also for general text strings.

## 4.2.10.2 Cautions and warnings

! This function returns a pointer to a static buffer area. Multiple calls may re–use the same buffer.

## 4.2.10.3 Formal description

```
const char *
error_package_name
(
    error_number_t base
);
```

### 4.2.10.3.1 Return value

This function returns a "`char *`" pointer to a static character buffer containing the pathname of a SLAPS product's *message catalog* as stored in the SLAPS repository.

### 4.2.10.3.2 Parameters

`base` This parameter is an *error base*, of type `error_number_t`. Actually, any *error number* will work; the low 16 bits are ignored. This parameter specifies the product for which the *message catalog's* pathname will be returned.

### 4.2.10.3.3 Special behavior

## 4.2.10.4 Examples

```
{
    nl_catd cd;

    setlocale(LC_ALL, "");
    cd = catopen(error_package_name(ERROR_EXAMPLE_BASE), 0);
    puts(catgets(cd, 2, 1, "The example program is starting."));

    /* ... */
```

## 4.2.10.5 Errors

This function will fail only if the indicated SLAPS product either does not exist or does not have message catalogs. In this case, it will return an invalid filename. Please note that this function does not verify that the file actually exists, can be accessed, or is in the proper format. It returns the name as stored internally in the SLAPS system.

Passing an invalid filename to `catopen` will merely result in that function returning an invalid catalog descriptor. Using this invalid catalog descriptor in `catgets` calls will merely result in the use of the default strings. Therefore, no explicit error checking is needed: the code example in 4.2.10.4 above will, if the catalog is unavailable, happily continue along using the default text.

## 4.2.10.6 References to related operations

See also,

- `catopen(3)`
- `catgets(3)`

### 4.2.11 `error_stack`

#### 4.2.11.1 Functional description

`error_stack` adds an *error message* onto an *error stack*, creating the *error stack* if it does not already exist.

#### 4.2.11.2 Cautions and warnings

! `error_stack` *is a C preprocessor macro.*

#### 4.2.11.3 Formal description

```
error_t
error_stack
(
    error_t e,
    error_number_t n,
    const char *m,
    ...
);
```

##### 4.2.11.3.1 Return value

`error_stack` returns an *error stack*, of type `error_t`. It never returns `error_ok`.

##### 4.2.11.3.2 Parameters

`e` This parameter is an *error stack*, of type `error_t`. It specifies the stack onto which the new message is to be placed. This parameter may be passed as zero, in which case a new *error stack* is created and the message becomes the bottommost message.

`n` This parameter is an *error number*, of type `error_number_t`. It specifies the error situation which this new message reports.

`m` This parameter points to a string, and is of type “`char *`.” This string is the *default error message text*. It may contain conversion specifiers in the style of `printf(3)`. The parameters to be converted follow.

... This function takes a variable number of arguments. The default error message text contains conversion specifiers which determine how these parameters, if any, are to be interpreted and stored. These parameters follow the default error message text parameter.

##### 4.2.11.3.3 Special behavior

`error_stack` always returns a non–`error_ok` *error stack,* but this never indicates that it itself encountered an error.

#### 4.2.11.4 Examples

```
{
    error_t e;

    e = error_stack(0, 0x12340100, "First error message.");
    e = error_stack(e, 0x12340101, "%s error message.", "second");
    e = error_stack(e, 0x12340102, "Message number %d.", 3);
    e = error_stack(e, 0x12340102, "Message number %d.", 4);
    e = error_stack(e, 0x12340103, "Last error message.");

    error_output(e);
    error_free(e);
}
```

#### 4.2.11.5 Errors

`error_stack` does not return an error to indicate that it had a problem. If it encounters difficulties (e.g., it cannot allocate memory), it copes (e.g., it trims information from the *error stack* (losing parameters and replacing intermediate messages with brief warnings)). It does not fail. It does, however, always return a non–`error_ok` *error stack.*

#### 4.2.11.6 References to related operations

See also,

- `error_output`
- `error_unix`
- `error_vstack`
- `Error_stack`
- `printf(3)`

## 4.2.12 error_text

### 4.2.12.1 Functional description

error_text creates a set of text strings that corresponds to an *error stack*. This is useful if a program wishes to display an error to a user without using an *error server*.

### 4.2.12.2 Cautions and warnings

! This routine allocates storage, the freeing of which is the responsibility of the user.

### 4.2.12.3 Formal description

```
error_t
error_text
(
    error_t e,
    char ***t
);
```

#### 4.2.12.3.1 Return value

This function returns an *error stack*. When successful, it returns error_ok.

#### 4.2.12.3.2 Parameters

e This parameter is an *error stack*. The text generated corresponds to this *error stack*.

t This is an output parameter. It points to an area into which this routine places a pointer to a null–terminated list of character string pointers.

#### 4.2.12.3.3 Special behavior

The second parameter of this routine is an output parameter. This is the only call in the *error library* with an output parameter.

### 4.2.12.4 Examples

```
error_t pslerror(error_t e)
{
    if( e )
    {
        char **text, **x;

        error_t e2 = error_text(e, &text);
        if( e2 != error_ok )
        {
            return error_stack(e2, ERROR_EXAMPLE_PSLERROR,
                "Cannot print an error stack at 0x%08x to stderr.", e);
        }

        for( x = text; *x; x++ )
        {
            fprintf(stderr, %s\n", *x);
            free(*x);
        }

        free(text);
    }

    return error_ok;
}
```

#### 4.2.12.5 Errors

`error_text` can fail for a few reasons, including an inability to allocate memory, or problems with the scratch file sometimes used for very long messages. In the case of an error, a non–`error_ok` *error stack* will be returned with the topmost *error number* equal to

- `ERROR_ERROR_TEXT`

#### 4.2.12.6 References to related operations

See also,

- `error_output`
- `Error_text`

## 4.2.13 error_top

### 4.2.13.1 Functional description

error_top returns the *error number* of the *topmost error message* of an *error stack.*

### 4.2.13.2 Cautions and warnings

! error_top *is a C preprocessor macro*

### 4.2.13.3 Formal description

```
error_number_t
error_top
(
    error_t e
);
```

#### 4.2.13.3.1 Return value

This macro returns an *error number*, of type error_number_t.

#### 4.2.13.3.2 Parameters

e This parameter is an *error stack*, of type error_t. It specifies the *error stack* of which the *error number* of the *topmost error message* is returned. This parameter is required, and there is no default.

#### 4.2.13.3.3 Special behavior

This macro returns a zero when its parameter is error_ok. The results are undefined if it is not passed a valid *error stack.*

### 4.2.13.4 Examples

```
{
    error_t e;
    error_number_t n;

    e = error_stack(NULL, 0x12340001, "Message one.");
    e = error_stack(e,    0x12340002, "Message two.");
    e = error_stack(e,    0x12340003, "Message three.");

    n = error_top(e);

    assert( n == 0x12340003 );

    error_free(e);
}
```

### 4.2.13.5 Errors

error_top does not return any errors.

### 4.2.13.6 References to related operations

See also,

- error_bottom
- error_nth
- Error_top

## 4.2.14 `error_unix`

### 4.2.14.1 Functional description

`error_unix` creates an *error stack* with an initial *error message* which reports that a *system call* has failed with a particular `errno` value.

### 4.2.14.2 Cautions and warnings

### 4.2.14.3 Formal description

```
error_t
error_unix
(
    const char *call
);
```

#### 4.2.14.3.1 Return value

This function returns an *error stack*, but never reports an error of its own. It never returns `error_ok`.

#### 4.2.14.3.2 Parameters

`call` This parameter is a pointer to a string identifying the failed system call.

#### 4.2.14.3.3 Special behavior

`error_unix` always returns a non–`error_ok` *error stack*, but this never indicates that it itself encountered an error.

### 4.2.14.4 Example

```
int
safe_read
(
    int d,
    char *buf,
    int nbytes
)
{
    int r = read(d, buf, nbytes);

    if( r == -1 )
    {
        error_t e = error_unix("read");
        e = error_stack(e, ERROR_EXAMPLE_READ,
            "Cannot read up to %d bytes from fd %d into a buffer at 0x%08x.",
            nbytes, d, buf);
        return e;
    }

    return r;
}
```

### 4.2.14.5 Errors

This routine calls `error_stack` internally, so see section 4.2.11.5.

#### 4.2.14.6 References to related operations

See also,

- `error_stack`
- `Error_unix`
- `perror(3)`

### 4.2.15 `error_vstack`

#### 4.2.15.1 Functional description

`error_vstack` is a version of `error_stack` which, instead of taking a variable number of arguments, takes a `stdarg` argument vector.

#### 4.2.15.2 Cautions and warnings

! `error_vstack` *is a C preprocessor macro.*

#### 4.2.15.3 Formal description

```
error_t
error_vstack
(
    error_t e,
    error_number_t n,
    const char *m,
    va_list ap
);
```

##### 4.2.15.3.1 Return value

`error_vstack` returns an *error stack*, of type `error_t`. It never returns `error_ok`.

##### 4.2.15.3.2 Parameters

`e` This parameter is an *error stack*, of type `error_t`. It specifies the stack onto which the new message is to be placed. This parameter may be passed as zero, in which case a new *error stack* is created and the message becomes the bottommost message.

`n` This parameter is an *error number*, of type `error_number_t.` It specifies the error situation which this new message reports.

`m` This parameter points to a string, and is of type "`char *`." This string is the *default error message text*. It may contain conversion specifiers in the style of `printf(3)`. The parameters to be converted follow.

`ap` This parameter is a variable argument list (see `stdarg(3)`). These arguments are the values which get converted by the *error message* text string.

##### 4.2.15.3.3 Special behavior

`error_stack` always returns a non–`error_ok` *error stack,* but this never indicates that it itself encountered an error.

#### 4.2.15.4 Examples

```
error_t
safe_execl
(
    char *path,
    ...
)
{
    va_list ap;
    char *args[MAXARGS], msgbuf[MAXARGS*8 + 32];
    int argno;
    error_t e;

    va_start(ap, path);
    while( args[argno++] = va_arg(ap, char *));
    va_end(ap);

    (void)execv(path, args);

    e = error_unix("execv");
    strcpy(msgbuf, "Cannot exec %s with parameters:");
    while( argno-- ) strcat(msgbuf, " 0x%08x");
    e = error_stack(e, ERROR_EXAMPLE_EXECL, msgbuf, ap);

    return e;
}
```

#### 4.2.15.5 Errors

`error_vstack` does not return an error to indicate that it had a problem. If it encounters difficulties (e.g., it cannot allocate memory), it copes (e.g., it trims information from the *error stack* (losing parameters and replacing intermediate messages with brief warnings)). It does not fail. It does, however, always return a non–`error_ok` *error stack*.

#### 4.2.15.6 References to related operations

See also,

- `error_stack`
- `Error_vstack`
- `stdarg(3)`
- `vprintf(3)`

## 4.2.16 `Error_bottom`

### 4.2.16.1 Functional description

`Error_bottom` returns the *error number* of the *bottommost error message* of the *standard error stack.*

### 4.2.16.2 Cautions and warnings

! `Error_bottom` *is a C preprocessor macro.*

### 4.2.16.3 Formal description

```
error_number_t
Error_bottom
(
    void
);
```

#### 4.2.16.3.1 Return value

This macro returns an *error number,* of type `error_number_t`.

#### 4.2.16.3.2 Parameters

This macro does not take any explicit parameters. It uses, but does not modify, the *standard error stack.*

#### 4.2.16.3.3 Special behavior

This macro returns a zero when the *standard error stack* is `error_ok`.

### 4.2.16.4 Examples

```
{
    error_number_t n;

    Error_stack(0x12340001, "Message one.");
    Error_stack(0x12340002, "Message two.");
    Error_stack(0x12340003, "Message three.");

    n = Error_bottom();

    assert( n == 0x12340001 );

    Error_free();
}
```

### 4.2.16.5 Errors

`Error_bottom` does not return any errors.

### 4.2.16.6 References to related operations

See also,

- `error_bottom`
- `Error_nth`
- `Error_top`

# 4.2.17 `Error_depth`

## 4.2.17.1 Functional description

`Error_depth` returns the number of *error messages* in the *standard error stack.*

## 4.2.17.2 Cautions and warnings

! `Error_depth` *is a C preprocessor macro.*

## 4.2.17.3 Formal description

```
int
Error_depth
(
    void
);
```

### 4.2.17.3.1 Return value

This macro returns an `int`.

### 4.2.17.3.2 Parameters

This macro does not take any explicit parameters. It uses, but does not modify, the *standard error stack.*

### 4.2.17.3.3 Special behavior

This macro returns a zero when the *standard error stack* is `error_ok`.

## 4.2.17.4 Examples

```
{
    int depth

    Error_stack(0x12340001, "Message one.");
    Error_stack(0x12340002, "Message two.");
    Error_stack(0x12340003, "Message three.");

    depth = Error_depth();

    assert( n == 3 );

    Error_free();
}
```

## 4.2.17.5 Errors

`error_depth` does not return any errors.

## 4.2.17.6 References to related operation

See also,

- `error_depth`
- `Error_nth`

## 4.2.18 `Error_export`

### 4.2.18.1 Functional description

`Error_export` sends the *standard error stack* over a file descriptor.

### 4.2.18.2 Cautions and warnings

! `Error_export` no longer clears the *standard error stack* to `error_ok` after it has been successfully exported.

### 4.2.18.3 Formal description

```
void
Error_export
(
    int fd
);
```

#### 4.2.18.3.1 Return value

This routine does not have an explicit return value. When unsuccessful, it places *error messages* on the *standard error stack* and raises an *exception.*

#### 4.2.18.3.2 Parameters

This function uses the *standard error stack.*

`fd` This parameter is a *file descriptor,* of type `int`. It specifies the socket, file, pipe, or other filesystem object to which the *standard error stack* is to be exported.

#### 4.2.18.3.3 Special behavior

The special *error stack* `error_ok` is a perfectly valid value for the *standard error stack*, and will be exported properly. This routine raises an exception upon failure.

### 4.2.18.4 Examples

```
try
{
    int p[2];
    Pipe(p);

    if( Fork() == 0 )
    {   /* child */
        try
            Error_stack(0x12340011, "Message one");
            Error_stack(0x12340012, "Message two");
            Error_export(p[1]);
        catchall:
            Error_output();
        endtry;
        exit(0);
    }
    else /* parent */
    {
        try
            Error_import(p[0]);
        catchall:
            Error_output();
        endtry;
    }
}
catchall:
    Error_stack(ERROR_EXAMPLE_EXIM, "Export/import example failed.");
    reraise;
endtry;
```

### 4.2.18.5 Errors

`Error_export` can fail for two reasons:

- `ERROR_ERROR_NO_MEMORY`: The routine cannot temporarily allocate enough memory to marshall the data.
- `ERROR_ERROR_WRITE`: An underlying call to `write(2)` failed.

In the case of an error, one or more *error messages* will be placed on the *standard error stack*, and an exception raised. The exception will be indicative of the bottommost problem; the topmost *error number* will be one of the two constants listed above.

### 4.2.18.6 References to related operations

See also,

- `error_export`
- `Error_fexport`
- `Error_import`
- `Error_output`

### 4.2.19 `Error_fexport`

#### 4.2.19.1 Functional description

`Error_fexport` writes the *standard error stack* to a file identified by a "`FILE *`" file pointer.

#### 4.2.19.2 Cautions and warnings

! `Error_fexport` no longer clears the *standard error stack* to `error_ok` after it has been successfully exported.

#### 4.2.19.3 Formal description

```
void
Error_fexport
(
    FILE *fp
);
```

##### 4.2.19.3.1 Return value

This routine does not have an explicit return value. When unsuccessful, it places *error messages* on the *standard error stack* and raises an *exception.*

##### 4.2.19.3.2 Parameters

This function uses the *standard error stack.*

`fp` This parameter is a *file pointer,* of type `FILE *`. It specifies the filesystem object to which the *standard error stack* is to be exported.

##### 4.2.19.3.3 Special behavior

The special *error stack* `error_ok` is a perfectly valid value for the *standard error stack*, and will be exported properly. This routine raises an exception upon failure.

#### 4.2.19.4 Examples

```
try
{
    FILE *fp;

    Error_stack(0x12340021, "First message.");
    Error_stack(0x12340022, "Second message.");

    Fopen("logfile.err", "w");
    Error_fexport(fp);
}
catchall:
    Error_stack(ERROR_EXAMPLE_FEXPORT, "Fexport example failed.");
    reraise;
endtry;
```

#### 4.2.19.5 Errors

`Error_fexport` can fail for two reasons:

- `ERROR_ERROR_NO_MEMORY`: The routine cannot temporarily allocate enough memory to marshall the data.
- `ERROR_ERROR_FWRITE`: An underlying call to `fwrite(3)` failed.

In the case of an error, one or more *error messages* will be placed on the *standard error stack*, and an exception raised. The exception will be indicative of the bottommost problem; the topmost *error number* will be one of the two constants listed above.

#### 4.2.19.6 References to related operations

- `error_fexport`
- `Error_export`
- `Error_fimport`

## 4.2.20 `Error_fimport`

### 4.2.20.1 Functional description

`Error_fimport` sets the *standard error stack* to a value read from a file identified by a "`FILE *`" file pointer

### 4.2.20.2 Cautions and warnings

! `Error_fimport` may set the *standard error stack* to a non–`error_ok` value either when it encounters an error and cannot function, or when it successfully reads a non–`error_ok` stack. When it succeeds, it returns; when it fails, it raises an exception.

### 4.2.20.3 Formal description

```
void
Error_fimport
(
    FILE *fp
);
```

#### 4.2.20.3.1 Return value

This function does not return a value. It sets the *standard error stack* to be either the *error stack* it read successfully, or to a stack indicating the error it encountered.

#### 4.2.20.3.2 Parameters

`fp` This parameter is a *file pointer,* of type `FILE *`. It specifies the filesystem object from which the *standard error stack* is to be imported.

#### 4.2.20.3.3 Special behavior

This routine raises an exception upon failure.

### 4.2.20.4 Examples

```
try
{
    FILE *fp = Fopen("logfile.err", "r");

    while( 1 )
    {
        Error_fimport(fp);
        feed_histograms(Error_getstack());
        Error_free();
    }
}
catchall:
    Error_stack(ERROR_EXAMPLE_FIMPORT, "Fimport example failed.");
    reraise;
endtry;
```

### 4.2.20.5 Errors

`Error_fimport` can fail for two reasons:

- `ERROR_ERROR_NO_MEMORY`: The routine cannot allocate enough memory to create the *error stack.*
- `ERROR_ERROR_FREAD`: An underlying call to `fread(3)` failed.

In the case of an error, one or more *error messages* will be placed on the *standard error stack*, and an exception raised. The exception will be indicative of the bottommost problem; the topmost *error number* will be one of the two constants listed above.

#### 4.2.20.6 References to related operations

- `error_fimport`
- `Error_fexport`
- `Error_import`

# 4.2.21 `Error_free`

## 4.2.21.1 Functional description

`Error_free` frees all allocated space associated with the *standard error stack*, and clears it back to `error_ok`.

## 4.2.21.2 Cautions and warnings

## 4.2.21.3 Formal description

```
void
Error_free
(
    void
)
```

### 4.2.21.3.1 Return value

This function does not return any value.

### 4.2.21.3.2 Parameters

This routine clears the *standard error stack.*

### 4.2.21.3.3 Special behavior

Freeing a *standard error stack* with value `error_ok` does not accomplish anything, but it is perfectly safe to do it.

## 4.2.21.4 Examples

```
try
    subroutine();
catch(ERROR_EXAMPLE_WARNING):
    Error_free();
    break;
catchall:
    Error_stack(ERROR_EXAMPLE_FAILED, "subroutine failed with more than a warning.");
    reraise;
endtry;
```

## 4.2.21.5 Errors

This function does not have any errors.

## 4.2.21.6 References to related operations

- `error_free`
- `error_stack`
- `Error_stack`

## 4.2.22 `Error_getstack`

### 4.2.22.1 Functional description

`Error_getstack` returns the *standard error stack.*

### 4.2.22.2 Cautions and warnings

! Be careful when manipulating the *error stack* returned by this routine. If you change it, the *standard error stack* may be invalid unless you reset it with `Error_setstack`. An invalid *standard error stack* may make many `Error_`*something* routines act in an undefined way.

### 4.2.22.3 Formal description

```
error_t
Error_getstack
(
    void
);
```

#### 4.2.22.3.1 Return value

This function returns an *error stack*. However, this routine does not report a failure.

#### 4.2.22.3.2 Parameters

This routine does not take any parameters. It uses, but does not modify, the *standard error stack.*

#### 4.2.22.3.3 Special behavior

If this routine fails internally (which can only happen if the POSIX threads support fails), it aborts.

### 4.2.22.4 Examples

```
try
{
    FILE *fp = Fopen("logfile.err", "r");

    while( 1 )
    {
        Error_fimport(fp);
        feed_histograms(Error_getstack());
        Error_free();
    }
}
catchall:
    Error_stack(ERROR_EXAMPLE_FIMPORT, "Fimport example failed.");
    reraise;
endtry;
```

### 4.2.22.5 Errors

This routine does not return any error. If the POSIX threads support fails, it aborts.

### 4.2.22.6 References to related operations

- `Error_setstack`

## 4.2.23 `Error_import`

### 4.2.23.1 Functional description

`Error_import` sets the *standard error stack* to a value read over a socket, pipe, or other file descriptor.

### 4.2.23.2 Cautions and warnings

! `Error_import` may set the *standard error stack* to a non–`error_ok` value either when it encounters an error and cannot function, or when it successfully reads a non–`error_ok` stack. When it succeeds, it returns; when it fails, it raises an exception.

### 4.2.23.3 Formal description

```
void
error_import
(
    int fd
);
```

#### 4.2.23.3.1 Return value

This function does not return a value, but it does set the *standard error stack.*

#### 4.2.23.3.2 Parameters

`fd` This parameter is a *file descriptor*, of type `int`. It specifies the socket, pipe, or other file descriptor over which the *error stack* is to be read.

#### 4.2.23.3.3 Special behavior

### 4.2.23.4 Examples

```
try
{
    int p[2];
    Pipe(p);

    if( Fork() == 0 )
    {   /* child */
        try
            Error_stack(0x12340011, ”Message one”);
            Error_stack(0x12340012, ”Message two”);
            Error_export(p[1]);
        catchall:
            Error_output();
        endtry;
        exit(0);
    }
    else /* parent */
    {
        try
            Error_import(p[0]);
        catchall:
            Error_output();
        endtry;
    }
}
catchall:
    Error_stack(ERROR_EXAMPLE_EXIM, ”Export/import example failed.”);
    reraise;
endtry;
```

#### 4.2.23.5 Errors

`Error_import` can fail for two reasons:

- `ERROR_ERROR_NO_MEMORY`: The routine cannot allocate enough memory to create the *error stack.*
- `ERROR_ERROR_READ`: An underlying call to `read(3)` failed.

In the case of an error, one or more *error messages* will be placed on the *standard error stack*, and an exception raised. The exception will be indicative of the bottommost problem; the topmost *error number* will be one of the two constants listed above.

#### 4.2.23.6 References to related operations

See also,

- `error_import`
- `Error_export`
- `Error_fimport`

## 4.2.24 `Error_nth`

### 4.2.24.1 Functional description

`Error_nth` returns the *error number* of one specified *error message* of the *standard error stack.*

### 4.2.24.2 Cautions and warnings

! `Error_nth` *is a C preprocessor macro.*

### 4.2.24.3 Formal description

```
error_number_t
Error_nth
(
    int n
);
```

#### 4.2.24.3.1 Return value

This macro returns an *error number,* of type `error_number_t`.

#### 4.2.24.3.2 Parameters

`n` This parameter is an index, of type `int`. It specifies the *error message* of which the *error number* is to be returned. Zero corresponds to the bottommost message, `(Error_depth()-1)` corresponds to the topmost.

#### 4.2.24.3.3 Special behavior

This macro returns a zero when the *standard error stack* is `error_ok`. The results are undefined if the index is out of the range `[0, Error_depth(e))`.

### 4.2.24.4 Examples

```
{
    error_number_t n;

    Error_stack(0x12340001, "Message one.");
    Error_stack(0x12340002, "Message two.");
    Error_stack(0x12340003, "Message three.");

    n = Error_nth(1);

    assert( n == 0x12340002 );

    Error_free();
}
```

### 4.2.24.5 Errors

`error_nth` does not return any errors.

### 4.2.24.6 References to related operations

See also,

- `error_nth`
- `Error_bottom`
- `Error_depth`
- `Error_top`

### 4.2.25 `Error_output`

#### 4.2.25.1 Functional description

`Error_output` communicates the *standard error stack* to an *error server,* if one is running on the same host. Usually, computers with displays that run programs interactively will run an *error server.*

If there is not an *error server* running locally, the behavior is not yet defined. The original design of *the SL Error System* called for this routine to transmit the *standard error stack* to a *general error server* in the control room, which would display the errors of important background servers on an *Alarm System*–like display. However, this is one of the subjects which is being reconsidered by the new *Error System* developers, and it will have to be clarified in a future revision of this document.

If all else fails, and the *standard error stack* cannot be sent to an *error server* for appropriate handling, the text of the *standard error stack* is sent to `stderr`. By definition, this routine takes full responsibility for reporting an error.

#### 4.2.25.2 Cautions and warnings

! The original definition of `Error_output` reset the *standard error stack* to `eor_ok` after successfully sending it. This was subsequently changed, so now `Error_output` does *not* reset it. If you do not have a further use for the *standard erorr stack* (such as logging it), remember to call `Error_free`.

#### 4.2.25.3 Formal description

```
void
Error_output
(
    void
);
```

##### 4.2.25.3.1 Return value

This function does not return anything.

##### 4.2.25.3.2 Parameters

This function does not take any parameters. It does use the *standard error stack.*

##### 4.2.25.3.3 Special behavior

At the time of writing, the *SL Error Server* is not yet operational. This routine currently uses the *SPS Error Server* instead. See SPS/AOP/OS/Note 46 for more details about the *SPS Error System.*

#### 4.2.25.4 Examples

```
{
    Error_stack(0x12340030, "Bottom message.");
    Error_stack(0x12340031, "Middle message.");
    Error_stack(0x12340032, "Top message.");

    Error_output();
    Error_free();
}
```

#### 4.2.25.5 Errors

This routine does not produce any errors.

#### 4.2.25.6 References to related operations

See also,

- `error_output`
- `Error_text`

### 4.2.26 `Error_setstack`

#### 4.2.26.1 Functional description

`Error_setstack` assigns its value to the *standard error stack.* This is useful when calling a library routine which returns an *error stack* in a program which uses exceptions.

#### 4.2.26.2 Cautions and warnings

! If the *standard error stack* already has a value when this is called, the old value is lost, but the space is not reclaimed. This *is* a feature. After calling this routine, further manipulation of the parameter may make the *standard error stack* invalid.

#### 4.2.26.3 Formal description

```
void
Error_setstack
(
    error_t e
);
```

##### 4.2.26.3.1 Return value

This routine does not return anything.

##### 4.2.26.3.2 Parameters

`e` This is the *error stack* which is to become the *standard error stack.*

##### 4.2.26.3.3 Special behavior

This routine changes the *standard error stack.* If this routine fails internally (which can only happen if the POSIX threads support fails), it aborts.

#### 4.2.26.4 Examples

```
void
Library_call()
{
    error_t e = library_call();
    Error_setstack(e);

    if( e != error_ok )
        raise( Error_bottom() );
    else
        return;
}
```

#### 4.2.26.5 Errors

This routine does not return any errors. If the POSIX threads support fails, it aborts.

#### 4.2.26.6 References to related operations

See also,

- `Error_getstack`

### 4.2.27 `Error_stack`

#### 4.2.27.1 Functional description

`Error_stack` adds an *error message* to the *standard error stack.*

#### 4.2.27.2 Cautions and warnings

! `Error_stack` *is a C preprocessor macro.*

#### 4.2.27.3 Formal description

```
void
Error_stack
(
    error_number_t n,
    const char *m,
    ...
);
```

##### 4.2.27.3.1 Return value

`Error_stack` does not return anything.

##### 4.2.27.3.2 Parameters

`n` This parameter is an *error number*, of type `error_number_t.` It specifies the error situation which this new message reports.

`m` This parameter points to a string, and is of type "`char *`." This string is the *default error message text*. It may contain conversion specifiers in the style of `printf(3)`. The parameters to be converted follow.

... This function takes a variable number of arguments. The default error message text contains conversion specifiers which determine how these parameters, if any, are to be interpreted and stored. These parameters follow the default error message text parameter.

##### 4.2.27.3.3 Special behavior

This routine changes the *standard error stack.*

#### 4.2.27.4 Examples

```
{
    Error_stack(0x12340100, "First error message.");
    Error_stack(0x12340101, "%s error message.", "second");
    Error_stack(0x12340102, "Message number %d.", 3);
    Error_stack(0x12340102, "Message number %d.", 4);
    Error_stack(0x12340103, "Last error message.");

    Error_output();
    Error_free();
}
```

#### 4.2.27.5 Errors

`Error_stack` does not return an error to indicate that it had a problem. If it encounters difficulties (e.g., it cannot allocate memory), it copes (e.g., it trims information from the *error stack* (losing parameters and replacing intermediate messages with brief warnings)). It does not fail.

#### 4.2.27.6 References to related operations

See also,

- `error_stack`
- `Error_output`
- `Error_unix`
- `Error_vstack`
- `printf(3)`

## 4.2.28 Error_top

### 4.2.28.1 Functional description

Error_top returns the *error number* of the *topmost error message* of the *standard error stack.*

### 4.2.28.2 Cautions and warnings

! Error_top *is a C preprocessor macro.*

### 4.2.28.3 Formal description

```
error_number_t
Error_top
(
    void
);
```

#### 4.2.28.3.1 Return value

This macro returns an *error number*, of type error_number_t.

#### 4.2.28.3.2 Parameters

This macro does not take any explicit parameters. It uses, but does not modify, the *standard error stack.*

#### 4.2.28.3.3 Special behavior

This macro returns a zero when the *standard error stack* is error_ok.

### 4.2.28.4 Examples

```
{
    error_number_t n;

    Error_stack(0x12340001, "Message one.");
    Error_stack(0x12340002, "Message two.");
    Error_stack(0x12340003, "Message three.");

    n = Error_top();

    assert( n == 0x12340003 );

    Error_free();
}
```

### 4.2.28.5 Errors

Error_top does not return any errors.

### 4.2.28.6 References to related operations

See also,

- error_top
- Error_bottom
- Error_nth

## 4.2.29 `Error_unix`

### 4.2.29.1 Functional description

`Error_unix` creates an *error message* which reports that a *system call* has failed with a particular `errno` value, and places it on the *standard error stack.*

### 4.2.29.2 Cautions and warnings

### 4.2.29.3 Formal description

```
void
Error_unix
(
    const char *call
);
```

#### 4.2.29.3.1 Return value

This function does not return anything.

#### 4.2.29.3.2 Parameters

`call`This parameter is a pointer to a string identifying the failed system call.

#### 4.2.29.3.3 Special behavior

This call modifies the *standard error stack.*

### 4.2.29.4 Example

```
int
Read
(
    int d,
    char *buf,
    int nbytes
)
{
    int r = read(d, buf, nbytes);

    if( r == -1 )
    {
        Error_unix("read");
        Error_stack(ERROR_EXAMPLE_READ,
            "Cannot read up to %d bytes from fd %d into a buffer at 0x%08x.",
            nbytes, d, buf);
        raise(ERROR_EXAMPLE_READ);
    }

    return r;
}
```

### 4.2.29.5 Errors

This routine calls `Error_stack` internally, so see section 4.2.26.5.

### 4.2.29.6 References to related operations

See also,

- `error_unix`
- `Error_stack`
- `perror(3)`

## 4.2.30 `Error_vstack`

### 4.2.30.1 Functional description

`Error_vstack` is a version of `Error_stack` which, instead of taking a variable number of arguments, takes a `stdarg` argument vector.

### 4.2.30.2 Cautions and warnings

! `Error_vstack` *is a C preprocessor macro.*

### 4.2.30.3 Formal description

```
void
error_vstack
(
    error_number_t n,
    const char *m,
    va_list ap
);
```

#### 4.2.30.3.1 Return value

`Error_vstack` does not return anything.

#### 4.2.30.3.2 Parameters

`n` This parameter is an *error number*, of type `error_number_t.` It specifies the error situation which this new message reports.

`m` This parameter points to a string, and is of type "`char *`." This string is the *default error message text*. It may contain conversion specifiers in the style of `printf(3)`. The parameters to be converted follow.

`ap` This parameter is a variable argument list (see `stdarg(3)`). These arguments are the values which get converted by the *error message* text string.

#### 4.2.30.3.3 Special behavior

This macro uses and changes the *standard error stack.*

### 4.2.30.4 Examples

```
void
Execl
(
    char *path,
    ...
)
{
    va_list ap;
    char *args[MAXARGS], msgbuf[MAXARGS*8 + 32];
    int argno;

    va_start(ap, path);
    while( args[argno++] = va_arg(ap, char *));
    va_end(ap);

    (void)execv(path, args);

    Error_unix("execv");
    strcpy(msgbuf, "Cannot exec %s with parameters:");
    while( argno-- ) strcat(msgbuf, " 0x%08x");
    Error_vstack(ERROR_EXAMPLE_EXECL, msgbuf, ap);

    raise(ERROR_EXAMPLE_EXECL);
}
```

### 4.2.30.5 Errors

`Error_vstack` does not return an error to indicate that it had a problem. If it encounters difficulties (e.g., it cannot allocate memory), it copes (e.g., it trims information from the *error stack* (losing parameters and replacing intermediate messages with brief warnings)). It does not fail.

### 4.2.30.6 References to related operations

See also,

- `error_vstack`
- `Error_stack`
- `stdarg(3)`
- `vprintf(3)`

# 4.3 Exceptions

## 4.3.1 General description

### 4.3.1.1 Functional description

Exceptions provide another way of changing the flow of control of a program. They are usually used to indicate an *exceptional* condition, i.e., an error.

Basically, a function or subroutine in exception–using code consists of two parts: the normal, main part, which performs the intended tasks; and the exception–handling, or error–handling, part. In "normal" coding, these are intertwined; with exceptions, they can be separated. This makes the common situation where any error requires a standard "back–out" or recovery set of steps easier to write and understand.

Some languages, like ADA, have a built–in exception mechanism. The syntax is such that any function or routine has the two parts. There is a proposed system for C++. C, however, does not have a built–in mechanism, but with its preprocessor macro facility one can be constructed. This system, written for use in and with the error system, is implemented with C preprocessor macros and one function call, and relies on the `sigsetjmp` and `siglongjmp` calls for the nonlocal transfer.

A block of code to be "protected" with an exception handler begins with the macro `try`. After the block, one can have zero or more blocks beginning with `catch(`*n*`):` which handle specific exception conditions. There may optionally be one block beginning with `catchall:` which catches any exception not already caught in that `try` block (this is similar to `case` *n*`:` and `default:`.) The section started with `try` then ends with `endtry`. Blocks may be nested, and do not have to correspond to function calls or subroutines.

An exception is "raised," which means an exceptional transfer of control is made to a corresponding `catch`, with the function call `raise(`*n*`)`. At this point, control jumps to the corresponding `catch(`*n*`)` or the `catchall` block of the latest `try/endty` region. If this latest region does not have the corresponding `catch(`*n*`)` or the `catchall` block, or if the code in the block executes the macro `reraise`, the same exception is re–raised to the next innermost `try/endtry` region. This continues until a region finally catches the exception, or the first region is reached. This latter case is an error, and will cause the program to abort with an "unhandled exception" message.

In a multithreaded environment, this exception handling system is thread–safe. Each thread has its own set of exception handlers. Any number of threads may be simultaneously raising exceptions with no inter–thread interference.

### 4.3.1.2 Cautions and warnings

! `try`, `endtry`, `catch()`, `catchall`, and `reraise` are all C preprocessor macros.

! Do not `return`, `goto`, or `continue` out of the main block. (These are fine *within* the block).

! This mechanism uses `sigsetjmp` and `siglongjmp`; see the warnings on their *man pages*. If these calls are implemented properly, it should be ok to `raise` an exception out of a signal handler.

! Some SysV versions of UNIX defined `raise()` to be a library call with which a program can send itself a signal (sigh). On these platforms, the <Syscalls.h> header file gets around this problem by redefining the SysV library call. In any case, the SysV functionality can easily be replaced with `kill` and `getpid`.

! This system is not in any way compatable with the exception system used in the first SL RPC system. The author considers this a feature.

! Do not use variables beginning with `exception`.

#### 4.3.1.3 Formal description

*See the individual sections for the formal description of each macro or routine.*

- `break`

A `break` statement within a `try/endtry` region which is not otherwise caught by a loop or `switch` statement within that region will transfer control to after the `endtry`. This is typically used in a `catch` block after successfully recovering from the exceptional situation. Also, since it is not safe to `return` out of the main block, it is used instead of a premature return.

- `continue`

A `continue` statement in a `catch` or `catchall` block which is not otherwise caught by a loop statement loops control back to the beginning of the `try` block. This is useful if the exception–handling code corrects some problem and wishes to re–try the main code. Do not `continue` out of a `try` block..

#### 4.3.1.4 Examples

```
try
    /* main code */
    /* main code */
    /* main code */
catch(1):
    /* code to handle exception #1 */
    break;
catch(2):
    /* code to handle exception #2 */
    break;
catchall:
    /* code to handle all exceptions except #1 and #2 */
    break;
endtry;
```

#### 4.3.1.5 Errors

It is possible for a `try`, `raise`, or `reraise` to encounter a system call error with `sigsetjmp` or `siglongjmp`; or on multithreaded platforms, with the “thread–specific storage” library calls. Since there is no way of recovering from these errors (and they are pathological problems anyway), they result in warning messages being issued to `stderr`, followed by a process abort.

If you encounter one of these errors, please let the author know.

#### 4.3.1.6 References to related operations

See also,

- `catch`
- `catchall`
- `endtry`
- `raise`
- `reraise`
- `try`

## 4.3.2 catch

### 4.3.2.1 Functional description

This serves as a label for a block of code within a section protected by an exception handler. When an exception is raised in the main block of the section, and the exception number matches the argument of the `catch` macro, control is non–locally transferred to the labelled block. There may be zero or more `catch` labels in a protected section, but each must have a unique argument.

### 4.3.2.2 Cautions and warnings

! `catch` is a C preprocessor macro.

### 4.3.2.3 Formal description

```
catch(n):
```

Semantically, `catch(`*n*`):` is a label, very similar to `case` *n*`:`. The argument is an *exception number.*

### 4.3.2.4 Examples

*See 4.3.1.4.*

### 4.3.2.5 Errors

On multithreaded platforms, it is technically possible for the `pthread_setspecific` call used internally to fail. A failure would mean that the system `pthread` data structures have become corrupted. If this call fails, the program will print a message to `stderr` and abort.

### 4.3.2.6 References to related operations

See also,

- `catchall`
- `endtry`
- `raise`
- `reraise`
- `try`

## 4.3.3 catchall

### 4.3.3.1 Functional description

This serves as a label for a block of code within a section protected by an exception handler. When an exception is raised in the main block of the section, and the exception number does not match the argument of any `catch` macro, control is non–locally transferred to the labelled block. There may be zero or one `catchall` labels in a protected section.

### 4.3.3.2 Cautions and warnings

! `catchall` is a C preprocessor macro.

### 4.3.3.3 Formal description

```
catchall:
```

Semantically, `catchall:` is a label, very similar to `default:`.

### 4.3.3.4 Examples

*See 4.3.1.4.*

### 4.3.3.5 Errors

On multithreaded platforms, it is technically possible for the `pthread_setspecific` call used internally to fail. A failure would mean that the system `pthread` data structures have become corrupted. If this call fails, the program will print a message to `stderr` and abort.

### 4.3.3.6 References to related operations

See also,

- `catch`
- `endtry`
- `raise`
- `reraise`
- `try`

### 4.3.4 `endtry`

#### 4.3.4.1 Functional description

This macro indicates the end of a section of code protected by an exception handler. When the main code of the section completes successfully, control continues after the `endtry`. Also, if any code within the protected section executes a `break` not otherwise caught, execution continues after the `endtry`. This macro is required at the end of every protected section.

#### 4.3.4.2 Cautions and warnings

- ! `endtry` is a C preprocessor macro.
- ! `endtry` must be followed by a semicolon.
- ! There must be one `endtry` for every `try`, and they must be on the same block level (like a `do/while`).

#### 4.3.4.3 Formal description

```
endtry;
```

Semantically, `endtry` is a statement; or rather, a `try` and `endtry` together with the code between them are a statement. `try` ... `endtry;` is semantically similar to `do` ... `while(`*e*`);`.

#### 4.3.4.4 Examples

*See 4.3.1.4.*

#### 4.3.4.5 Errors

On multithreaded platforms, it is technically possible for the `pthread_setspecific` or `pthread_getspecific` calls used internally to fail. A failure would mean that the system `pthread` data structures have become corrupted. If this call fails, the program will print a message to `stderr` and abort.

#### 4.3.4.6 References to related operations

See also,

- `catch`
- `catchall`
- `raise`
- `reraise`
- `try`

### 4.3.5 raise

#### 4.3.5.1 Functional description

This function initaiates a non–local transfer of control to the block of code, labelled by a `catchall` or a `catch` corresponding to the argument of the `raise`, of the innermost (most recently entered) protected section. If the innermost protected section does not have a matching `catch` or `catchall`, the exception is re–raised to the next innermost one, etc. If there is no matching `catch` or `catchall` at all, the program aborts. There may be zero or more `raise` calls in a protected block.

#### 4.3.5.2 Cautions and warnings

! `raise` is a function.

#### 4.3.5.3 Formal description

```
void
raise
(
    exception_t n
);
```

#### 4.3.5.4 Examples

*See 4.3.1.4.*

#### 4.3.5.5 Errors

If an exception is raised, but not caught, the process prints an error message to `stderr` and aborts. On multithreaded platforms, it is technically possible for the `pthread_getspecific` calls used internally to fail. A failure would mean that the system `pthread` data structures have become corrupted. If this call fails, the program will print a message to `stderr` and abort.

#### 4.3.5.6 References to related operations

See also,

- `catch`
- `catchall`
- `raise`
- `reraise`
- `try`

## 4.3.6 reraise

### 4.3.6.1 Functional description

Within a `catch` or `catchall` block of code, `reraise` explicitly re–raises the exception which caused that block to execute to the next protected section. This macro may not be called from within the main block of a protected section, nor outside a protected section.

### 4.3.6.2 Cautions and warnings

! `reraise` is a C preprocessor macro.

### 4.3.6.3 Formal description

```
reraise;
```

Semantically, `reraise` is a statement, like `break` or `continue`.

### 4.3.6.4 Examples

*See 4.3.1.4.*

### 4.3.6.5 Errors

If an exception is raised, but not caught, the process prints an error message to `stderr` and aborts. On multithreaded platforms, it is technically possible for the `pthread_getspecific` calls used internally to fail. A failure would mean that the system `pthread` data structures have become corrupted. If this call fails, the program will print a message to `stderr` and abort.

### 4.3.6.6 References to related operations

See also,

- `catch`
- `catchall`
- `endtry`
- `raise`
- `try`

## 4.3.7 try

### 4.3.7.1 Functional description

This macro begins a section of code protected by an exception handler. The block of code between the `try` and the next `catch`, `catchall`, or `endtry` is the main block of the section. Inside this main block, a `raise` will cause a non–local transfer of control to a matching `catch` or `catchall` block. This macro is required at the beginning of every protected section.

### 4.3.7.2 Cautions and warnings

! `try` is a C preprocessor macro.

### 4.3.7.3 Formal description

```
try
```

Semantically, `try` is a statement, like `do`. Or rather, it, its `endtry;`, and the code in–between are a statement; like `do`, some code, and `while();`.

### 4.3.7.4 Examples

*See 4.3.1.4.*

### 4.3.7.5 Errors

It is possible for the `sigsetjmp` system call, or on multithreaded platforms the thread–specific storage calls, to fail. Although all of the exception macros and routines suffer from this problem, only with `try` is a failure likely: with the others, the only failure mode is if the internal system data becomes corrupted, but `try` will (the first time a `try` is reached in a program) try to allocate a thread–specific storage "key." Depending on the POSIX threads implementation, it is conceivable that a program just on the edge of resource exhaustion might fail inside a `try`. In any case, an error in `try`, like any other exception keyword, will cause an error message to be printed to `stderr` and the program to abort. The author would like to know if this ever actually happens.

### 4.3.7.6 References to related operations

See also,

- `catch`
- `catchall`
- `endtry`
- `raise`
- `reraise`

# 4.4 Product requirements

## 4.4.1 Obtain an *error base*

Request an *error base* from the SLAPS administrator. The SLAPS administrator will edit the file `/user/slaps/rep/products/error_lib/lib/error_bases.h` to add a C preprocessor macro for your product. This will be of the form,

`#define ERROR_`*product_name*`_BASE      0x`*nnnn*`0000`

where *product_name* is the name of your product, and 0x*nnnn*0000 is a new, unique base. Also, if the product has message catalogs, the SLAPS administrator will record the product's *error base* and the message catalog location.

## 4.4.2 Define errors

Every product which provides data to another product is required by SLAPS to have a *public include file*. Usually, these products are servers or subroutine libraries. If these products might return an *error stack*, then the public include file must contain mnemonic error number macros of the form,

`#define ERROR_`*product_name_actual_error*`   (ERROR_`*product_name*`_BASE + `*n*`)`

where *product_name* is as above, *actual_error* is some brief name for the error (often the name of a failing subroutine or request), and *n* is a small positive integer unique in this situation.

Standalone or user–interface programs are not required to have a public include file; however, it is suggested that a private include file be created with mnemonic error macros.

Usually this list of errors is created one–by–one as the program is written.

## 4.4.3 Create message catalogs

If the developer wishes to write an internationalized product, with multilingual messages, he must create a *message catalog* for each supported language. Following standard internationalization guidelines, the filenames for these catalog sources should be named,

*la*_*CO*`.`*charset*`.msg`

where *la* is the ISO 639 standard two–letter language identifier, *CO* is the ISO 3166 two–letter country identifier, and *charset* specifies the character set used. See the following tables:

| *language identifier* | *language* |
|---|---|
| `en` | English |
| `fr` | French |
| `de` | German |
| `es` | Spanish |
| *etc.* | |

| *country identifier* | *country* |
|---|---|
| `US` | United States |
| `FR` | France |
| `CH` | Switzerland |
| `IL` | Israel |
| *etc.* | |

Currently, the only character set supported on the SL Control System workstations is "`iso88591`," for ISO 8859–1, which supports western European languages.

## 4.5 Standard wrapper libraries

The wrappers for the standard system and library calls are not going to be individually documented. Their description is best left to the *man pages* of the underlying calls, and their prototypes are easily derived.

## 4.6 Errors and RPCs

At the time of writing, the interaction between RPCs and the SL Error System has not been completely worked out. This section is deferred until this work has been completed.

## 4.7 Errors and SL–EQUIP

At the time of writing, the interaction between the SL–EQUIP package and the SL Error System has not been completely worked out. This section is deferred until this work has been completed.

## 4.8 SL Error Server reference

At the time of writing, the SL Error Server has not been released. The SPS Error Server is being used for the time being, so see *some appropriate reference.*

# A. Error Messages

## A.1. Exceptions:

`ERROR_EXCEPTION_BASE` `0x00000000`

The exception package is defined to have an *error base* of zero; but this base, and these errors, are never actually seen by the error system or anything else. They are only used internally. The low parts of the numbers are used to look up messages in the exception product's message catalogs.

`ERROR_EXCEPTION_UNHANDLED` `0x00000001`

This is the error code for unhandled exceptions. An unhandled exception is an exception that has been raised but never caught. It is a grave programming error.

`ERROR_EXCEPTION_ABORT` `0x00000002`

This is the error code which indicates the program is about to abort due to a failure of the exception system. This error never happens by itself; it is the result of one of the other errors on this page.

`ERROR_EXCEPTION_PTHREAD_ONCE` `0x00000003`

On multithreaded platforms, each process thread has its own system of exceptions. On singlethreaded platforms, the exception stack can be pointed to by an external variable, but on multithreaded platoforms thread–specific storage must be used. To use this, a storage key must be created *once* in the program. The library call `pthread_once` is used to ensure that only one key is created. If this call fails, the exception system cannot initialize itself, and the program cannot run. This error code indicates this failure. A failure of this library call probably indicates that the system thread data has been corrupted.

`ERROR_EXCEPTION_PTHREAD_GETSPECIFIC` `0x00000004`

On multithreaded platforms, each process thread has its own system of exceptions, the list of which must be pointed to by a pointer in thread–specific storage. The library call `pthread_getspecific`, which gets the pointer from thread–specific storage, may fail. A failure of this library call probably indicates that the system thread data has been corrupted. If this call fails, the exception system cannot initialize itself, and the program cannot run. This error code indicates this failure.

`ERROR_EXCEPTION_PTHREAD_SETSPECIFIC` `0x00000005`

On multithreaded platforms, each process thread has its own system of exceptions, the list of which must be pointed to by a pointer in thread–specific storage. The library call `pthread_getspecific`, which gets the pointer from thread–specific storage, may fail. A failure of this library call probably indicates that the system thread data has been corrupted. If this call fails, the exception system cannot initialize itself, and the program cannot run. This error code indicates this failure.

`ERROR_EXCEPTION_PTHREAD_KEYCREATE` `0x00000006`

On multithreaded platforms, each process thread has its own system of exceptions, the list of which must be pointed to by a pointer in thread–specific storage. To use this, a storage key must be created *once* in the program. This library call `pthread_keycreate` is used to create this key. This call may fail, either because system thread information has been corrupted, or on some implementations because of resource exhaustion. This error code indicates that this library call has failed.

## A.2. The SL Error Library

`ERROR_ERROR_BASE` `0x00010000`

`ERROR_ERROR_NO_MEMORY` `0x00010001`

This error indicates that no more memory can be allocated by the error system. The error system tries very hard to get around this problem. If an error cannot be stacked because of memory exhaustion, parameters may be trimmed, default messages may be truncated, and *error messages* in the middle of the *error stack* may be compressed into one warning message. There is a limited amount of emergency storage, about enough to hold a message stating that memory was exhausted, and that that is probably why the error occurred. This is the error code of the "out of memory" message. This is also used in `error_text`; in this usage, it may be seen by a user.

`ERROR_ERROR_INVALID_MESSAGE` `0x00010002`

This is never used as an *error number;* merely as an *exception.* It is used internally, and should never be seen outside the SL Error System. It indicates that the default text of an *error message* has invalid *conversion specifiers,* and therefore the parameters cannot be properly interpreted. The visible result of this situation is an error report message without parameters.

`ERROR_ERROR_INTERNAL_EXCEPTION` `0x00010003`

This error indicates that the error system has detected an unhandled exception internal to itself. It indicates an error system internal error. This error is never seen by the user; if it occurs, the program aborts.

`ERROR_ERROR_UNKNOWN_UNIX_ERROR` `0x00010004`

The `error_unix` and `Error_unix` calls create an *error message* to report the failure of a system call. System calls traditionally return an invalid value and set `errno` to a value which indicates the problem. These calls use this `errno` value to create the proper *error message.* If either of these calls finds `errno` to be out of the documented range, this error code is used instead of an ERROR_*platform*_ERRNO_*errno* one.

`ERROR_UNIX_ERROR_DEFAULT_MSG` `0x00010005`

If the `error_unix` or `Error_unix` calls cannot find the proper system call error text, they substitute the text corresponding to this number. This number is never actually used as an *error number.*

`ERROR_ERROR_INTRO` `0x00010006`

This number is never used as an *error number;* it is used by `error_text` to find the introductory error report text in the message catalogs.

`ERROR_ERROR_NO_ERROR` `0x00010007`

This is *not* the value of `error_ok` (`error_ok` is zero). This is used by `error_text` to find the "No error" text in the message catalogs. It is never used as an *error number.*

`ERROR_ERROR_TEXT` 0x00010008

This is a genuine *error number* which indicates that `error_text` cannot create a text version of an *error stack.* There should be further messages explaining why.

`ERROR_SAFE_PTHREAD_ONCE` 0x00010009

On singlethreaded platforms, the *standard error stack* is an external variable. On multithreaded platforms, since each thread has its own *standard error stack*, thread–specific storage must be used. The key for this storage must be created *once* in the lifetime of the process, and the library call `pthread_once` is used to ensure this unicity. If this routine fails, this is the *error number* of the error which reports this fact. Keep in mind that in some circumstances, the error system cannot use itself to report errors, and this responsibility cannot in general be passed back to the application: one purpose of this package is so that the developer does not have to worry about errors while handling errors. Therefore, if this routine fails at a bad time, the program may send a message to `stderr` and abort. Granted, the failure of this routine probably indicates that internal system information has become irrecoverably corrupted.

`ERROR_SAFE_PTHREAD_SETSPECIFIC` 0x0001000A

On multithreaded platforms, the thread–specific *standard error stacks* must be pointed to from thread–specific storage. The library call `pthread_setspecific`, which is used to store the pointers, can conceivably fail (usually because system information has become corrupted). If this call fails, this error is reported. If this failure occurs at an inopportune circumstance, the program may send a message to `stderr` and abort.

`ERROR_SAFE_PTHREAD_KEYCREATE` 0x0001000B

On multithreaded platforms, the thread–specific *standard error stacks* must be pointed to from thread–specific storage. The key for this storage must be created once in the lifetime of the process, and the library call `pthread_keycreate` is used to create it. If this call fails, this error is reported. If this failure occurs at an inopportune circumstance, the program may send a message to `stderr` and abort.

`ERROR_SAFE_PTHREAD_GETSPECIFIC` 0x0001000C

On multithreaded platforms, the thread–specific *standard error stacks* must be pointed to from thread–specific storage. The library call `pthread_getspecific`, which is used to retrieve the pointers, can conceivably fail (usually because system information has become corrupted). If this call fails, this error is reported. If this failure occurs at an inopportune circumstance, the program may send a message to `stderr` and abort.

`ERROR_ERROR_FPRINTF` 0x0001000D

The `error_text` routine uses a temporary file when converting the parameters of error messages with long error texts. `fprintf` is used to access this file. If this access fails, this error will be on the returned *error stack.*

`ERROR_ERROR_FPUTC` 0x0001000E

The `error_text` routine uses a temporary file when converting the parameters of error messages with long error texts. `fputc` is used to access this file. If this access fails, this error will be on the returned *error stack.*

`ERROR_ERROR_FPUTS` 0x0001000F

The `error_text` routine uses a temporary file when converting the parameters of error messages with long error texts. `fputs` is used to access this file. If this access fails, this error will be on the returned *error stack.*

ERROR_ERROR_FREAD 0x00010010

The `error_text` routine uses a temporary file when converting the parameters of error messages with long error texts. `fputs` is used to access this file. Also, the `error_fimport` and `Error_fimport` routines use `fread`. If any of these accesses fail, this error will be on the resulting *error stack.*

ERROR_ERROR_FSEEK 0x00010011

The `error_text` routine uses a temporary file when converting the parameters of error messages with long error texts. `fseek` is used to access this file. If this access fails, this error will be on the returned *error stack.*

ERROR_ERROR_FTELL 0x00010012

The `error_text` routine uses a temporary file when converting the parameters of error messages with long error texts. `ftell` is used to access this file. If this access fails, this error will be on the returned *error stack.*

ERROR_ERROR_LOCALTIME 0x00010013

The `error_text` routine calls `localtime` to convert an *error message's* timestamp from the internal UNIX format to local time for display. If this call fails, this error will be returned on the *error stack.*

ERROR_ERROR_GET_STACK 0x00010014

On multithreaded platforms, the thread–specific *standard error stacks* must be pointed to from thread–specific storage. The `Error_getstack` call gets this value from there, and this call is used by other routines. If `Error_getstack` fails, this error will be used to find the text which is sent to `stderr` just before the program aborts.

ERROR_ERROR_SET_STACK 0x00010015

On multithreaded platforms, the thread–specific *standard error stacks* must be pointed to from thread–specific storage. The `Error_setstack` call stores this value there, and this call is used by other routines. If `Error_setstack` fails, this error will be used to find the text which is sent to `stderr` just before the program aborts.

ERROR_ERROR_WRITE 0x00010016

`Error_export` and `error_export` use `write` internally. If this `write` call fails, this error is placed on the resultant *error stack.*

ERROR_ERROR_READ 0x00010017

`Error_import` and `error_import` use `read` internally. If this `read` call fails, this error is placed on the resultant *error stack.*

ERROR_ERROR_FWRITE 0x00010018

`Error_fexport` and `error_fexport` use `fwrite` internally. If this call fails, this error is placed on the resultant *error stack.*

# A.3. Errno

The SL Error System errors which correspond to the UNIX `errno` values will not be individually documented. For each SL Control System platform, the *error base* for `errno` errors is listed below.

| | |
|---|---|
| ERROR_APOLLO_ERRNO_BASE | 0x00020000 |
| ERROR_LYNX_ERROR_BASE | 0x00030000 |
| ERROR_HPUX_ERROR_BASE | 0x000A0000 |
| ERROR_ULTRIX_ERROR_BASE | 0x000B0000 |

(The *error bases* are different because the constants vary between platforms.)

For each platform, the header file /user/slaps/rep/include/error_errno.h defines constants for the SL Error System version of each errno value, e.g.,

```
#define ERROR_ERRNO_EPERM  (ERROR_APOLLO_ERRNO_BASE + EPERM)
#define ERROR_ERRNO_ENOENT (ERROR_APOLLO_ERRNO_BASE + ENOENT)
etc.
```

Therefore, a routine which calls a standard wrapper call, e.g. Read, can check for specific errno values (e.g., ERROR_ERRNO_EBADF) in a portable way.

# A.4. Standard wrapper libraries

Each wrapper for a system call which sets errno will have an *error message* on the stack corresponding to that errno value. In addition, all wrappers have a unique error code. The bases are as follows:

| | |
|---|---|
| ERROR_APOLLO_BSD_SYSCALL_BASE | 0x00040000 |
| ERROR_APOLLO_BSD_LIBCALL_BASE | 0x00050000 |
| ERROR_APOLLO_SYSV_SYSCALL_BASE | 0x00060000 |
| ERROR_APOLLO_SYSV_LIBCALL_BASE | 0x00070000 |
| ERROR_LYNX_SYSCALL_BASE | 0x00080000 |
| ERROR_LYNX_LIBCALL_BASE | 0x00090000 |
| ERROR_HPUX_SYSCALL_BASE | 0x000C0000 |
| ERROR_HPUX_LIBCALL_BASE | 0x000D0000 |
| ERROR_ULTRIX_SYSCALL_BASE | 0x000E0000 |
| ERROR_ULTRIX_LIBCALL_BASE | 0x000F0000 |

The mnemonic name for each wrapper's failure error message is like one of the bases above, but with "BASE" replaced by the capitalized call name, e.g.,

ERROR_APOLLO_BSD_SYSCALL_READ or
ERROR_HPUX_LIBCALL_MALLOC.

# B. Acronyms

| | |
|---|---|
| **ARCH** | Asynchronous Return Channel |
| **DOMAIN** | Distributed Operating Multi–Access Interactive Network |
| **ECA** | Equipment Control Assembly |
| **IPC** | Inter–Process Communication |
| **LEP** | Large Electron–Positron Collider |
| **RPC** | Remote Procedure Call |
| **SL** | SPS/LEP |
| **SL/CO** | SL Controls |
| **SL/OP** | SL Operations |
| **SLES** | SL Error System |
| **SPS** | Super Proton Synchrotron |
| **SUM** | Software User Manual |
| **VMS** | VAX Mangement System |

# C. Glossary

### Alarm

Another term for a fault state; see *fault state*.

### Alarm console

An active user interface to the CERN Alarm System which provides information about active fault states.

### Alarm System, CERN

The processes, programs, libraries, and communications channels dedicated to the managing of fault states. This system was originally designed for the SL Control System; it is now used throughout CERN.

### Apollo

The first company to sell computer workstations; also, the general term for their workstations.

### Application programs

Software at or near the topmost level of the SL Control System; usually those programs run on the workstations in the control rooms.

### Asynchronous Return Channel (ARCH)

An early error facility for LEP, which was not used much.

### Background task

A computer process not directly connected to a user interface.

### Bottom

Of an error stack: the error message relating to the earliest cause of the problem.

### Catch

To receive and handle an exception.

### Character set

A set of glyphs and the numbers by which computers identify them. There are several standard sets; the SL Control System mostly uses ISO 8859–1, which supports several popular western European languages.

### Controls (SL/CO)

The group of people within the SL division responsible for the SL Control System.

### Conversion specifier

In the C programming language, a standard syntax used to specify the conversion of program variables into a textual format suitable for presentation to people. See the on–line documentation for `printf(3)` for a full description.

### Cron (**`cron`**)

A UNIX facility for periodically performing a task.

### Depth

The number of error messages on an error stack.

### Distributed Operating Multi–Access Interactive Network (DOMAIN)

The operating system used by Apollo workstations.

### Document Change Record

One or more pages in a document which for each document version provide a detailed list of the changes made. This software user manual contains a document change record as required by ESA standard PSS–05–05, though it isn't very detailed.

### Document Status Sheet

One or more pages in a document which list the current and all previous versions of the document. For each version there is specified the release date, and a brief description of the changes made since the previous version. This software user manual contains a document status sheet as required by ESA standard PSS–05–05; see that reference for more information.

### Environment variable

In UNIX, a datum used to store information about the environment in which a process is running. See the UNIX documentation for a further discussion.

### Equipment

The LEP and SPS hardware controlled by the SL Control System.

### Equipment Control Assembley (ECA)

A type of low–level computer in the SL Control System directly attached to equipment.

### Equipment server

A process (or its program) called by the SL–EQUIP system to control a specific piece of equipment.

### Errno (**`errno`**)

A variable used by UNIX system calls to return a limited amount of error information.

### Error

The inability of a piece of software to perform its task correctly.

## Error base

An error number with the low sixteen bits zero. An error base identifies a product within the SL Control System; a product's error numbers are small positive biases from its error base.

## Error library

The library of software stored in SLAPS which supports the use of the SL Error System. The SLAPS environment variable which identifies the error library is `$(SL_ERRORLIB)`.

## Error message

An erro rnumber and its accompanying textual description. The message may contain conversion specifiers for the inclusion of parameters which vary at runtime.

## Error number

A 32–bit integer which uniquely specifies a particular error. The high 16 bits specify a particular software product, and the low 16 bits specify a particular error within that product. This combination means a reported error number can be traced to a specific line of code in a specific product in the entire SL control system.

## Error object

An error stack. Although the SL Error System was not written in an object–oriented language, and was intended to be used in "conventional" software, it was designed with object–oriented goals in mind. One can think of abstract error objects being passed around between the various SL Control System elements, accreting error messages as they go.

## Error server

A program which receives error objects from other software for presentation to the appropriate people. This term also refers to a process running this program.

## Error stack

An ordered set of error messages, which describes the cause–and–effect chain of events which make up a single failure.

## European Center for Nuclear Research (CERN)

An international laboratory for high energy physics research, near Geneva Switzerland.

## Exception

In software engineering, an exception is a nonlocal or nonstandard flow of control resulting from an "exceptional" condition, e.g., an error or undefined operation.

## Exception handler

A unit of software code dedicated to recovering from the condition that caused an exception.

### Export

The output from a process of an object, particularly one with a nontrivial internal structure.

### Fault state

The description of a potential situation that requires attention; particularly, of a situation whose state is monitored by the CERN Alarm System. A fault state can be either active or inactive; when active, this means there is a problem which requires attention. In the SL Control System, active fault states usually indicate either an equipment malfunction or the detection of a dangerous situation (e.g., fire or flooding).

### File descriptor

The UNIX term for a small nonnegative integer which identifies a valid communication stream (usually a file) within a process.

### File pointer (`FILE *`)

The standard C implementation type which is used to refer to a valid communication stream (usually a file), together with its buffering facilities, within a process.

### General Error Server

The name for a specific instance of an error server intended to report errors from background tasks.

### Header file

In the C programming language, a file containing software source code which is included by the C preprocessor in another file when that other file is compiled.

### HP–UX

The operating system used by some Hewlett–Packard computers, including many of the ones used in the SL Control System.

### Import

1) The input to a process of an object, particularly one with nontrivial internal structure, 2) the connection of an RPC client with a particular server.

### Init (`init`)

A UNIX facility for automatically starting processes upon the startup of a computer.

### Internationalization (I18N)

The practise or process of writing or converting software to support multiple languages and presentation formats.

### Internet daemon (`inetd`)

A UNIX facility for initiating a process to handle an incoming network connection.

### Interprocess Communication (IPC)

The communication of information between separate processes, possibly on separate computers.

### Large Electron Positron Collider (LEP)

A large lepton collider at CERN. At 27 kilometers in circumference; it was at the time of writing the largest accelerator in the world.

### Library

A collection of compiled software that may be used by a program.

### Localization

The process of tailoring an internationalized software package for use in a particular locale; often this largely consists of translating the text used in the user interface.

### Macro

In the C programming language, a construct which is expanded by the preprocessor into actual C code according to a preprocessor definition.

### Magic number

A nondeterministically chosen value intended to uniquely identify something, usually in a setting where there exists no mechanism for an assurredly unique identification.

### Manual page (`man page`)

A entry of the on–line UNIX documentation.

### Message catalog

A text file, or its compiled equivalent, containing all messages in a specific language for a particular piece of internationalized software.

### Multithreaded

Software which is written to use multiple threads of control within a single process.

### Null device (`/dev/null`)

On UNIX computers, a pseudo–device used to discard information.

### Operations (SL/OP)

Members of the SL Operations group, who are responsible for the running of the SPS and LEP.

### Pipe

In UNIX, a unidirectional communications channel between two processes on the same computer.

### Preprocessor

In the C programming language, there are some simple compilation control directives defined as ‘‘preprocessor’’ directives. Traditionally, a separate program, called the preprocessor, processed these directives before the actual C compiler was invoked. These directives include conditional compilation, the inclusion of other files, and macro expansion.

## Process

In UNIX, an executing instance of a program.

## Product

A set of related software stored in SLAPS as a single unit.

## Prototype

In the C programming language, a mention of a procedure or function defined elsewhere, included so the compiler knows the types of any arguments or return value used.

## Public include file

A C header file relating to a public library or server which contains the information required by software to use the library or server. The SLAPS product checkin mechanism stores a copy of a product's public include file in a well known directory.

## Public library

Any library in SLAPS intended for general use by the SL programming community. There is usually a SLAPS environment variable for every public library.

## Raise

To initiate an exception.

## Remote Procedure Call (RPC)

An IPC mechanism which provides for communication between processes (possibly on different computers) using the function/procedure paradigm.

## SL (SPS/LEP)

The division within CERN responsible for the SPS and LEP.

## SL Control System

The set of computers, networks, and software used to control the hardware of the SPS and LEP.

## SL–EQUIP

The SL Control System software product used to access much of the equipment within the SPS and LEP.

## SLAPS

The repository for SL Control System application software, and the software management tools and procedures associated with it.

## SLAPS Administrator

A member of the SL/CO Applications section holding the responsability for SLAPS.

## Snapshot

A copy of a file as it existed at some point in the past, used as an example of the file's format, but not necessarily containing fully up–to–date information.

## Socket

A bidirectional communications channel between two processes, usually on separate computers; also, the processes' endpoints of such a communications channel.

## Standard error (`stderr`)

In UNIX, the output stream of a process intended to be used for the reporting of errors.

## Standard error stack

In the SL Error System, a set of error stacks kept in thread–specific storage for use by programs which take advantage of the SL Error System's exception handling facilities; also, a particular thread's member of that set.

## Super Proton Synchrotron (SPS)

A large synchrotron accelerator at CERN. At the time of writing, it was the second–largest accelerator at CERN.

## System call

An operating system gate between a user process and the system kernel.

## Thread

A flow of execution control within a process. Modern operating systems can support multithreaded programs, or ones which have more than one thread of control within the same process.

## Thread–safe

Software which may safely be used in or by multithreaded programs.

## Thread–specific storage

In multithreaded C programs, data storage which is external to the program block but private to a specific thread.

## Throw

A synonm for raise.

## Top

Of an error stack: the error message relating to the most recent effect of the problem.

## Type

A specific defined format for the storing of data.

## VAX Management System (VMS)

An operating system written by Digital Equipment Corporation for their VAX and AXP computers.

## Wrapper (wrapper function)

A software function or procedure which does little more than call another function or procedure (that which it ‘‘is wrapped around’’) and convert the inputs and/or outputs to a more useful form.

# D. Index

*To be written.*

# E. Files

## E.1. Snapshots

This section contains snapshots of several files, both from the SL Error System and from selected example client programs. They are intended merely to serve as examples or illustrations. Some of these files have been renamed, reorganized, and/or edited since the initial release; but the comments have not always been updated.

/user/slaps/rep/include/error_bases.h

```
/*
 * bases.h
 *
 * Version 1.0.1 : Frederick Roeber : 30 June 1993
 *
 * This file contains the error base definitions.
 *
 */

/*
 * numbers should be less than 0x8000000
 */

#define ERROR_EXCEPTION_BASE           0x00000000   /* Exception handler error space */
#define ERROR_ERROR_BASE               0x00010000   /* Error handler errors */
#define ERROR_APOLLO_ERRNO_BASE        0x00020000   /* Standard numbers for errno's */
#define ERROR_LYNX_ERRNO_BASE          0x00030000   /* Standard numbers for errno's */
#define ERROR_APOLLO_BSD_SYSCALL_BASE  0x00040000   /* Exception-handler wrapper errors */
#define ERROR_APOLLO_BSD_LIBCALL_BASE  0x00050000   /* Exception-handler wrapper errors */
#define ERROR_APOLLO_SYSV_SYSCALL_BASE 0x00060000   /* Exception-handler wrapper errors */
#define ERROR_APOLLO_SYSV_LIBCALL_BASE 0x00070000   /* Exception-handler wrapper errors */
#define ERROR_LYNX_SYSCALL_BASE        0x00080000   /* Exception-handler wrapper errors */
#define ERROR_LYNX_LIBCALL_BASE        0x00090000   /* Exception-handler wrapper errors */
#define ERROR_HPUX_ERRNO_BASE          0x000A0000   /* Standard numbers for errno's */
#define ERROR_ULTRIX_ERRNO_BASE        0x000B0000   /* Standard numbers for errno's */
#define ERROR_HPUX_SYSCALL_BASE        0x000C0000   /* Exception-handler wrapper errors */
#define ERROR_HPUX_LIBCALL_BASE        0x000D0000   /* Exception-handler wrapper errors */
#define ERROR_ULTRIX_SYSCALL_BASE      0x000E0000   /* Exception-handler wrapper errors */
#define ERROR_ULTRIX_LIBCALL_BASE      0x000F0000   /* Exception-handler wrapper errors */
#define ERROR_ORACLE_BASE              0x00100000   /* Bias for Oracle error numbers */

#define ERROR_RAW_TELETEXT_BASE        0x10010000   /* Frederick Roeber */
#define ERROR_TELETEXT_BASE            0x10020000   /* Frederick Roeber */
#define ERROR_MFST_BASE                0x10030000   /* Frederick Roeber */

#define ERROR_MSRV_BASE                0x10040000   /* Eugenia */
#define ERROR_LSRVSGN_BASE             0x10050000   /* Eugenia */
#define ERROR_DBM_BASE                 0x10060000   /* Eugenia */
#define ERROR_GREYSHELL_BASE           0x10070000   /* Eugenia */

#define ERROR_OCI_BASE                 0x10080000   /* Frederick Roeber */
#define ERROR_DBDVX_BASE               0x10090000   /* Frederick Roeber */
#define ERROR_DVX_BASE                 0x100A0000   /* Frederick Roeber */
```

/user/slaps/rep/products/error_lib/lib/error_errors.h

```
#define ERROR_ERROR_NO_MEMORY               (ERROR_ERROR_BASE + 0x0001)
#define ERROR_ERROR_INVALID_MESSAGE         (ERROR_ERROR_BASE + 0x0002)
#define ERROR_ERROR_INTERNAL_EXCEPTION      (ERROR_ERROR_BASE + 0x0003)
#define ERROR_ERROR_UNKNOWN_UNIX_ERROR      (ERROR_ERROR_BASE + 0x0004)
#define ERROR_UNIX_ERROR_DEFAULT_MSG        (ERROR_ERROR_BASE + 0x0005)
#define ERROR_ERROR_INTRO                   (ERROR_ERROR_BASE + 0x0006)
#define ERROR_ERROR_NO_ERROR                (ERROR_ERROR_BASE + 0x0007)
#define ERROR_ERROR_TEXT                    (ERROR_ERROR_BASE + 0x0008)
#define ERROR_SAFE_PTHREAD_ONCE             (ERROR_ERROR_BASE + 0x0009)
#define ERROR_SAFE_PTHREAD_SETSPECIFIC      (ERROR_ERROR_BASE + 0x000a)
#define ERROR_SAFE_PTHREAD_KEYCREATE        (ERROR_ERROR_BASE + 0x000b)
#define ERROR_SAFE_PTHREAD_GETSPECIFIC      (ERROR_ERROR_BASE + 0x000c)
#define ERROR_ERROR_FPRINTF                 (ERROR_ERROR_BASE + 0x000d)
#define ERROR_ERROR_FPUTC                   (ERROR_ERROR_BASE + 0x000e)
#define ERROR_ERROR_FPUTS                   (ERROR_ERROR_BASE + 0x000f)
#define ERROR_ERROR_FREAD                   (ERROR_ERROR_BASE + 0x0010)
#define ERROR_ERROR_FSEEK                   (ERROR_ERROR_BASE + 0x0011)
#define ERROR_ERROR_FTELL                   (ERROR_ERROR_BASE + 0x0012)
#define ERROR_ERROR_LOCALTIME               (ERROR_ERROR_BASE + 0x0013)
#define ERROR_ERROR_GET_STACK               (ERROR_ERROR_BASE + 0x0014)
#define ERROR_ERROR_SET_STACK               (ERROR_ERROR_BASE + 0x0015)
#define ERROR_ERROR_WRITE                   (ERROR_ERROR_BASE + 0x0016)
#define ERROR_ERROR_READ                    (ERROR_ERROR_BASE + 0x0017)
#define ERROR_ERROR_FWRITE                  (ERROR_ERROR_BASE + 0x0018)
```

/user/slaps/rep/products/error_lib/lib/en_US.iso88591.msg

```
$
$ error/error/en_US.iso88591.msg
$
$ Version 1.0 : Frederick Roeber : 22 July 1992
$
$ This is the error system message catalog source file
$
$set 1 Error handling system errors
1 Error handling system error: out of memory
2 Error handling system error: supplied error message has an invalid format.
3 Error handling system error: unhandled internal exception
4 Error handling system error: unknown Unix errno value
5 Unix error %d (%s) in %s call.
6 Error number 0x%x at line %d of file %s:
7 No error.
8 Cannot convert error object at 0x%x to text.
9 Error handling system error: call to pthread_once failed
10 Error handling system error: call to pthread_setspecific failed
11 Error handling system error: call to pthread_keycreate failed
12 Error handling system error: call to pthread_getspecific failed
13 Error handling system error: call to fprintf failed
14 Error handling system error: call to fputc failed
15 Error handling system error: call to fputs failed
16 Error handling system error: call to fread failed
17 Error handling system error: call to fseek failed
18 Error handling system error: call to ftell failed
19 Error handling system error: call to localtime failed
20 Error handling system error: cannot get error stack
21 Error handling system error: cannot set error stack
22 Error handling system error: call to write failed
23 Error handling system error: call to read failed
24 Error handling system error: call to fwrite failed
```

/user/slaps/rep/products/error_lib/lib/fr_FR.iso88591.msg

```
$
$ error_lib/lib/fr_FR.iso88591.msg
$
$ Version 1.1 : Frédérick Roeber : 28 Juillet 1993
$
$ Ceci est le fichier des messages en français pour le système de traitement
$ des erreurs.
$
$set 1 Erreurs dans le traitment des erreurs.
1 Traitment des erreurs: pas de mémoire
2 Traitment des erreurs: message est invalide.
3 Traitment des erreurs: exception interne non traitée
4 Traitment des erreurs: valeur d'«errno» est inconnue.
5 Erreur unix nombre %d («%s») dans la procédure «%s».
6 Erreur nombre 0x%x à la ligne %d du fichier «%s»:
7 Pas d'erreur.
8 Ne peut pas transformer l'objet d'erreur à 0x%x en texte.
9 Traitment des erreurs: appel à «pthread_once» a échoué.
10 Traitment des erreurs: appel à «pthread_setspecific» a échoué.
11 Traitment des erreurs: appel à «pthread_keycreate» a échoué.
12 Traitment des erreurs: appel à «pthread_getspecific» a échoué.
13 Traitment des erreurs: appel à «fprintf» a échoué.
14 Traitment des erreurs: appel à «fputc» a échoué.
15 Traitment des erreurs: appel à «fputs» a échoué.
16 Traitment des erreurs: appel à «fread» a échoué.
17 Traitment des erreurs: appel à «fseek» a échoué.
18 Traitment des erreurs: appel à «ftell» a échoué.
19 Traitment des erreurs: appel à «localtime» a échoué.
20 Traitment des erreurs: Ne peut pas obtenir la pile d'erreur.
21 Traitment des erreurs: Ne peut pas mettre la pile d'erreur.
22 Traitment des erreurs: appel à «write» a échoué.
23 Traitment des erreurs: appel à «read» a échoué.
24 Traitment des erreurs: appel à «fwrite» a échoué.
```

/user/slaps/rep/products/teletext/teletext.h

```
/*
 * teletext/teletext.h
 *
 * Version 1.0 : Frederick Roeber : 7 September 1992
 *
 * This file is the public include file for the teletext daemon.
 *
 */

#include "error_bases.h"    /* for ERROR_TELETEXT_BASE */

#define ERROR_TELETEXT_OPEN_RAW_TELETEXT     (ERROR_TELETEXT_BASE +  1)
#define ERROR_TELETEXT_HE_HUNG_UP            (ERROR_TELETEXT_BASE +  2)
#define ERROR_TELETEXT_SAY                   (ERROR_TELETEXT_BASE +  3)
#define ERROR_TELETEXT_DUMP_PAGE             (ERROR_TELETEXT_BASE +  4)
#define ERROR_TELETEXT_READ_PAGE             (ERROR_TELETEXT_BASE +  5)
#define ERROR_TELETEXT_HARDWARE_ERROR        (ERROR_TELETEXT_BASE +  6)
#define ERROR_TELETEXT_UNEXPECTED_REPLY      (ERROR_TELETEXT_BASE +  7)
#define ERROR_TELETEXT_BAD_RAW               (ERROR_TELETEXT_BASE +  8)
#define ERROR_TELETEXT_BAD_COMMAND           (ERROR_TELETEXT_BASE +  9)
#define ERROR_TELETEXT_UNKNOWN_COMMAND       (ERROR_TELETEXT_BASE + 10)
    etc...
#define ERROR_TELETEXT_MAP                   (ERROR_TELETEXT_BASE + 37)
#define ERROR_TELETEXT_SIZESCREEN            (ERROR_TELETEXT_BASE + 38)
#define ERROR_TELETEXT_BELL                  (ERROR_TELETEXT_BASE + 39)
#define ERROR_TELETEXT_COMMAND_A             (ERROR_TELETEXT_BASE + 40)
```

/user/slaps/rep/products/teletext/en_US.iso88591.msg

```
$
$ teletext/en_US.iso88591.msg
$
$ Version 1.1 : Frederick Roeber : 21 June 1993
$
$ This is the American English message catalog for the teletext program.
$
$set 1 Teletext program errors
1 Could not open raw_teletext server on %s.
2 Raw teletext daemon disconnected from socket %d.
3 Cannot say something at 0x%x to raw teletext daemon on fd %d.
4 Cannot dump a the cooked version of a raw page.
5 Could not read a page from the client.
6 Hardware error.
7 Unexpected reply from hardware: '%c'
8 Cannot understand raw page from hardware.
9 Badly formed command.
10 Unknown command '%s'
    etc...
37 Cannot map cooked page data to raw data.
38 Cannot force screen to proper size.
39 Timeout (%d seconds)
40 Cannot list all active pages.
$set 2 (numbered) server responses
1 001 Error follows
10 010 Teletext daemon version %s ready\n
11 011 Teletext daemon version %s -- error follows\n
40 040 Errors will be sent as error objects.
50 050 Errors will be sent in text form.
60 060 Hello!  Now using English.
    etc...
```

# E.2. SLAPS Resources

The following files are installed in SLAPS in support of the SL Error System:

- `$(SL_INCLUDE)/error_bases.h`

This file defines the error bases.

- `$(SL_INCLUDE)/error_errno.h`

This file defines the mnemonic error number names for the error messages which correspond to `errno` values (e.g., "`ERROR_ERRNO_EIO`").

- `$(SL_INCLUDE)/error_handler.h`

This is the main header file which must be included to use the SL Error System

- `$(SL_INCLUDE)/exception.h`

This is the main header file which must be included to use the exception system.

- `$(SL_INCLUDE)/Error_handler.h`

This is the main header file which must be included to use both the error and exception systems. This file itself includes the previous two files.

- `$(SL_INCLUDE)/Libcalls.h`

This is the main header file which must be included to use the standard library call wrappers. It defines all of the mnemonic error number names, and prototypes all of the wrapper calls.

- `$(SL_INCLUDE)/Syscalls.h`

This is the main header file which must be included to use the standard system call wrappers. It defines all of the mnemonic error number names, and prototypes all of the wrapper calls.

- `$(SL_EXCEPTIONLIB)`
  `/user/slaps/rep/lib/`*arch*`/libexception_$(ISP).a`

Any program which uses exceptions must be linked with this library. `$(ISP)` refers to the machine architecture.

- `$(SL_ERRORLIB)`
  `/user/slaps/rep/lib/`*arch*`/liberror_$(ISP).a`
  `/user/slaps/rep/lib/`*arch*`/libexception_$(ISP).a`
  `/user/slaps/rep/lib/`*arch*`/Libcalls.a`
  `/user/slaps/rep/lib/`*arch*`/Syscalls.a`

Any program which uses the SL Error System must be linked with this set of libraries. The exception library mentioned above is included in this list. Because of the current circular dependencies in the libraries, this must be included *twice* on the link editor command line. *Arch* refers to the machine architecture.