# A Discipline of Error Handling

*Doug Moen*

## ABSTRACT

In the UNIX world, exception handling mechanisms for error handling are often discussed, but seldom applied. This paper describes a disciplined approach to error handling that was refined over a 3-year period during the development of a medium-large (200K line) toolkit written in C under UNIX. We describe both a portable exception handling system, written in C, and a methodology for using it which encompasses coding style, documentation, and testing issues.

### Introduction

The C language, as defined by Kernighan and Richey and by the ANSI C standard, is rather weak on error handling. The only standard error handling facilities provided are the global variable errno and the convention that certain functions (such as malloc, etc.) return a distinguished value when they fail, possibly setting errno.

This is not a very good basis for error handling.

It isn't good for application programmers, because explicitly checking the return values of functions that might fail, and propagating the error, is a lot of work, and clutters up code. Even conscientious programmers have been known to write programs that fail to check the return value of every call to printf. As a result, there are a non-trivial number of C programs in existence which fail to properly report error conditions [Darwin 85].

The standard approach to error handling isn't good for library implementors, either. One problem is that the existing set of error numbers is not extensible; thus, you can't use the standard functions perror(), strerror(), etc., with locally defined error codes. Another problem is that a single integer (errno) does not really contain enough information to completely describe an error. Usually there is additional contextual information (such as the name of the file that couldn't be opened, the number of bytes that were successfully written before an error occurred, the line number on which the error was detected, etc.) that needs to be associated with an error.

We faced these problems when we set out to build the EMS image processing toolkit in 1989. We were building a large library of functions for building image processing applications, and we wanted our library to support the construction of robust applications. We wanted our library to support good error handling. So we defined a comprehensive approach to good error handling, and wrote a small library of functions to support our error handling discipline.

There is more to good error handling than simply language or library support for raising and catching exceptions. In this paper, I will explain what errors are, describe the design issues in representing and reporting errors, and discuss how our error handling approach affects coding style, documentation and testing.

### Two Kinds Of Errors

EMS distinguishes two kinds of errors that can be detected by library functions: faults and failures.

A fault condition is the failure of an assertion or sanity check. By definition, a fault always indicates the presence of a bug in a program. Faults checking is not part of the contract between the function and its caller, and faults are reported by aborting the program. Some kinds of fault checking (e.g., comprehensive data structure integrity checks) are expensive to perform, but are useful during debugging. Because client code is not allowed to depend on the existence of fault checks, expensive fault checks can be conditionally compiled based on a DEBUG option, without affecting the correctness of any program.

A failure is an abnormal but anticipated condition such as resource exhaustion, permission denied, or a syntax error, which prevents the function from carrying out its job. Failures differ from faults in that failure reports are part of the contract between the function and its caller. Failures are reported by reporting an error back to the caller. An out of memory condition that causes malloc() to return NULL is a simple example of a failure.

Sometimes it is difficult to decide if a particular condition (e.g., an illegal argument value) should be classified as a fault or a failure. My rule of thumb is that it should be possible, using the library, to write programs that never generate fault conditions under any circumstances. Suppose that the exceptional condition is the detection of a syntax error in an input file, or in a character string that might have originated from outside of the program. In this case, the condition should be classified as a failure, rather than as a fault. If it were classified as a fault, then a program written to avoid generating faults is obliged to scan the input beforehand to ensure the absence of syntax errors.

### Reporting Faults

In EMS, faults are reported by calling the function fatal() with a printf-style argument list. fatal() aborts the program by calling a handler function registered using at_fatal(). If no handler function is registered, or if the registered handler function returns to its caller, then fatal() prints a message to stderr and calls abort() to obtain a core dump.

The decision to report faults by terminating the program was a controversial one. It was made for the following reasons:

- During a development cycle, the best way for a program to report a fault is to immediately dump core. The core file can then be used for post-mortem debugging.
- If a function signals a fault by reporting an error to its caller, then this error report becomes a de facto part of the function's contract with its caller, and it becomes possible to write code that depends on intercepting fault reports. We don't want fault reports to be part of a function's interface, because checking for faults can be expensive, and we would like to have the option of turn off fault checking using compile-time options, in order to speed up the code. Turning off fault checking should not change the semantics of a correct program.
- Assertions and sanity checks are easier to code, and are more likely to be used, if developers don't have to worry about cleaning up after a failed assertion.

An unfortunate consequence of aborting a program when a fault is detected is that we don't support fault tolerant programs of the sort that try to keep running after a fault has been detected (e.g., see [Meyer 88] and [Randell 75]). This isn't quite as bad as it sounds, because EMS has two provisions for supporting fault tolerance. First, programs can register a handler to save their state when fatal() is called. Second, EMS has facilities for automatically restarting crashed servers. Thus, an application that consists of a network of processes can be made to keep running even when individual components crash.

### Reporting Failures

The EMS failure reporting system is based on the following principles:

1. Error propagation. In a typical application, errors are detected at a low level (for example, in a library function), and handled at a higher level (for example, in an application program which calls the library). Only the higher level code knows how to handle the error; this might be to print an error message on the terminal, display an error window, or terminate the program. It is inappropriate for low level code to take these actions. Therefore, when an error is detected, a description of the error is propagated from the low level code, where it is detected, to the high level code (higher in the call stack) where it is handled.
2. Error values. In the traditional C approach to error handling, errors are described by a single small integer. This is inadequate; if the high level error handler needs to display an error message to the user, then it usually needs more than just the type of error (e.g., "syntax error"): it usually needs some information about the context in which the error occurred (e.g., file name + line number). In our system, errors are described by an Error structure which contains both an error id (which is an integer) and character string data which can be used to print an informative message.
3. Exception handling. In the traditional C approach to error handling, a function returns a distinguished value (such as NULL or -1) when it gets an error, and sets the global variable errno with an error id. It is the responsibility of the caller to check for an error return status, and either handle the error, or propagate it up the call stack. Although this approach is simple, it is also error prone: it is very easy for a programmer to be lazy, and omit checks for error return codes. If these checks are omitted, the program may go wrong in a catastrophic way when an error occurs. Our solution to this problem is to raise an exception when an error occurs. When a function raises an exception, it immediately terminates, its caller terminates, and so forth, until an exception handler is found. If no exception handler can be found anywhere on the call stack, the program terminates with an error message. Under this scheme, a programmer must take positive action to prevent his function from being terminated by an error. The worst that can happen if he forgets to check for errors is that, at the library level, temporarily allocated resources may not be freed, and at the application level, the program may terminate with a message.

### Error IDs

The most important component of an error description is an error 'id', which identifies the type of error that occurred. Error ids are used by error handlers to distinguish different types of errors. If the error id has a short string representation, then it can also be printed out as part of the error message, and used as a key by the end-user to look up a verbose description of the error in external documentation.

Any system for defining error ids needs to deal with two issues. First, it should be possible for programmers to define new error ids without editing a master table somewhere, and without conflicting with error ids defined by other libraries. Second, it should be possible to define sets of related error ids so that error handlers can check error ids for membership in a group, as an alternative to enumerating a long list of error ids that are to be handled in the same way.

The UNIX/ANSI C system of error ids (as defined by <errno.h> and sys_errlist) is not extensible, and provides no support for grouping.

Programming languages with built-in exception handling systems provide the ability to define new error ids as a matter of course, but not every such language provides a way to define groups of errors. In both ANSI C++ and in Common Lisp, it is possible to organize error types in trees or DAGs using single and multiple inheritance [Ellis 90, Steele 90].

In the exception handling system devised by Allman and Been [Allman 85], error ids are character strings, and error handlers can use glob-style pattern matching on error ids.

In EMS, we chose a system similar to that used by ANSI C and UNIX. Error ids are represented by integers, which means that error handlers can use switch statements to distinguish between different errors. When an error description is printed, the error id is used as an index into a table of message strings.

In order to support extensibility, we support multiple error tables. Each error id is a 32 bit integer with a 19 bit table id, and a 13 bit offset, which is an index into the table. The table id is computed from a table name: this is a string of between one and four lowercase letters which is converted to an integer using a variant of base 26 encoding.

Each error table is maintained as a text file that defines 4 records for each error:

- An error name, which is a C identifier like ENOENT or E_SYNTAX. The error name is used to #define a constant. In addition, it is printed by err_print_x for the benefit of both users and programmers. Users can use the error name to look up additional information about the error in externally provided documentation.
- A severity level, which is one of the constants ERR_FAULT, ERR_FAIL or ERR_RETRY. ERR_FAULT denotes a fault in an external program or subsystem, ERR_FAIL denotes a permanent failure, and ERR_RETRY indicates a temporary condition that will clear up by itself with no intervention.
- A short, one line message, analogous to the messages in sys_errlist. This message is printed by err_print_x().
- A verbose description of the error, typically one paragraph in length. This is incorporated into external documentation for errors.

This text file is processed to generate a .h file, a .c file, and a documentation file. The .h file contains one #defined constant for each error id, plus an external declaration for a global variable of type ErrTab_t[], representing the error table. The .c file contains the definition of the error table, which contains, for each error id, the message string, the severity level, and the error name, represented as a string.

By convention, in an error table named "foo", each error id has the prefix "EFOO_", the error table variable is called "foo_errs", and the header file is called "foo_errs.h".

System error codes, as obtained from the header file <errno.h> or the global variable errno, can be converted into valid error ids by casting them to type long. The corresponding error table is called sys_errs. All system errors are arbitrarily assigned the severity level ERR_FAIL.

The only provision that EMS has for defining groups of error ids are the fixed set of error severity levels. Although this has proven adequate for our needs, it is not very flexible.

## Error Values

When a function fails, it is responsible for conveying a description of the error to its caller. This description can be used in several different ways: it might be interpreted by an error handler which needs to take different actions on different errors, it might be used to construct an error message which will be displayed to a human user, and it might be transmitted to another process on the network (as when a server notifies a client of an error).

Since the error description might be interpreted by an error handler, it needs to contain an error id, plus any additional contextual information that might be needed by the error handler, such as the number of bytes that were successfully written before the error, the line number on which the error occurred, etc. Since the error description might be used to construct an error message, it needs to contain all of the information necessary to make this possible. Finally, since the error description might be transmitted to another host on the network, it needs to be represented in such a way that it can be transformed into a machine-independent byte stream for transmission purposes, then reconstituted by the recipient.

Let's consider the case of constructing an error message. Error messages are seen by two classes of people: end users, who don't understand the internals of the program that failed, and gurus/implementors who do.

For end users, the message should be phrased in high level terms, and provide enough information so that it is possible for the user to determine a corrective course of action. In practice, this means a one-line message containing a phrase describing the problem, the context in which the error occurred (e.g., file name, line number, etc.), and an error name which can be used as a key to look up a more complete description of the error in external documentation.

For gurus and implementors, the message should contain additional information about what went wrong at the implementation level of the program, because this information might be needed for debugging, or in order to fix a problem somewhere in the system. This lower level information can be generated naturally, as a consequence of the way that error descriptions are generated. When an error is detected, a description of the error is passed down the call stack through one or more levels of function calls until it is finally handled. Now, consider that each function is obligated to present an error interface which makes sense in terms of the abstraction that it implements. When an error description is forwarded through an abstraction boundary, it is sometimes necessary to reinterpret the error in terms of the current level of abstraction, which means supplying a new error id, and new contextual information. The lower level error description which is being supplanted need not be thrown away; this low level description of the error might be of use to gurus and implementors when it is presented in an error message.

Putting all of these requirements together, we find that an error description should consist of a stack of one or more error interpretations. The interpretation on the top of the stack is a high-level description of the error, while the interpretation at the bottom of the stack describes the error in terms of the function in which the error was first detected. Each error interpretation contains an error id, plus additional contextual information. We need operations on error descriptions for printing or displaying them as human readable messages, and for transmitting them across the network to other hosts (which may have different byte ordering, different floating point representations, etc.).

In EMS, error descriptions are represented by values of type Error_t. An Error_t contains a stack of error interpretations. Each interpretation contains an error id, the error name represented as a string, the short message associated with the error id, the error severity level, and a character string containing contextual information which is generated at run time, when the error is detected.

Note that the contextual information associated with each interpretation is represented as a single character string. This is a good representation for printing error messages and for transmitting error descriptions across the network, but it is not a convenient representation for error handlers which wish to access and interpret the contextual information. In fact, the current implementation of EMS provides no programmatic way to access the contextual information at all! Although this limitation has not caused any problems so far, I think it should be fixed.

The stack within an Error_t has a fixed maximum size of 6 entries, and there is a fixed amount of space for storing character string information. If you try to push more than 6 interpretations onto the stack, then the bottom interpretations are thrown away. If you try to store an oversize context string in an interpretation, then it will be truncated. We chose to use fixed sized arrays for representing Error_t's because we wanted to avoid the use of dynamically allocated storage. We did not want to run out of heap space while attempting to report errors in a low memory situation, and we did not want the additional complication of having to explicitly free the storage associated with Error_t's within an error handler. Instead, we use automatic storage for all Error_t's. In practice, the stack size limitation is not a problem, and neither is the occasional truncation of context strings.

### Operations On Error Values

When an error is first detected, an error value is initialized with a single error interpretation, consisting of an error id, the associated error name, severity level and short message, and an optional context string. If the error were printed at this point, it would look like the line in Figure 1, below.

An error value can be initialized to contain a single interpretation by calling err_set() (see Figure 2, below).

```
context string: short message (error name)
```

**Figure 1:** Error message format

```
void err_set( Error_t *err, ErrTab_t *tbl, long id, char *fmt, ... )
```

**Figure 2:** Prototypical `err_set` call

The fmt argument is the beginning of a printf argument list, which is used to construct the context string. See Figure 3 for an example.

When an error value passes through an abstraction boundary, it may be appropriate to push a new interpretation onto the stack which describes the error in higher level terms. This is done by calling err_push(). Alternatively, instead of pushing a new interpretation onto the stack, you can choose to push an annotation by calling err_note(). An annotation is a character string which is added to the top level error interpretation without changing the error id. Annotations are used to add information to to the error message seen by a user, without changing the error description from the point of view of an error handler. For example, the sequence of calls in Figure 4 would create an error description which would be printed as shown in Figure 5.

For completeness, we also supply err_clear(), which initializes an error description to an empty stack, and err_pop(), which pops the top level interpretation, so that the error id underneath can be accessed.

An error handler which needs to distinguish between different types of errors can query an error using the functions err_id() and err_level(). These functions return the 32 bit error id and the error severity level, respectively, of the top level interpretation within an error. The severity level is one of the following three constants:

```
#define ERR_RETRY 1
#define ERR_FAIL  2
#define ERR_FAULT 3
```

These constants are ordered by severity so that < and > tests can be used on them.

An error can be printed by calling err_print_x(). This function prefixes each line of output by the program name and a colon. If it is desired to display an error message within a dialog box, then you can call err_string_x() to obtain a character string representation of the error message, with embedded newlines, but without the program name prefixes.

Errors can be written to and read from a network connection in machine independent binary format by calling err_write_x() and err_read_x(). An unresolved problem with these functions is that UNIX error numbers change their interpretation from one machine to another.

### Internationalization

In multilingual environments, it is necessary to ensure that error messages are displayed in the correct natural language. Of course, error message strings are not the only strings that need to be translated, and it is appropriate to regard error handling and internationalization as orthogonal problems that deserve separate and independent solutions.

Different operating environments provide different solutions to the problem of internationalization. In the Macintosh and Windows environments, character strings that need to be translated are stored in resource files which are shipped with the application; the buyer must purchase a copy of the application which has been translated to the desired language. In the OSF environment, the NLS package supports a run-time choice of several different natural languages by setting the LANG environment variable; once again, strings containing natural language are stored separate from the programs themselves.

In the EMS toolkit, we have a high-level abstraction for string translation which can map cleanly onto the facilities provided by all of the above environments. The construct

```
XSD(string-literal)
```

is replaced by string-literal on systems that don't support internationalization, and is replaced by a function call which returns a pointer to the translated

```
Error_t err;
err_set(&err, sys_errs, (long)ENOMEM, "Couldn't allocate %d bytes", size);
```

**Figure 3**: Creating an error value

```
Error_t err;
err_set(&err, util_errs, E_EOF, "File descriptor %d", fd);
err_push(&err, util_errs, E_CORRUPT, "");
err_note(&err, "Error while reading file \"%s\"", filename);
```

**Figure 4**: Creating another error value

```
Error while reading file "foo"
> Corrupted data file (E_CORRUPT)
> File descriptor 3: Unexpected end of file (E_EOF)
```

**Figure 5**: A printed error value

string on systems that do. In other words,

```
XSD("Hello")
```

returns a pointer to the string ‘‘Bonjour’’ if compiled in an environment that supports internationalization, and the current language is French. The construct

```
XSI(string-literal)
```

returns a static initializer for a structure of type XS_t. The function xs_pgets() takes a pointer to an XS_t, and returns a pointer to the translated string. The XS package is implemented using a modified C preprocessor which maps string literals onto the appropriate magic incantations for fetching the translated version of that string from the appropriate resource file or database.

Thus, internationalization of error messages within the EMS environment becomes a trivial problem. The short error messages within each statically initialized ErrTab_t variable have type XS_t, and are initialized using the XSI macro; this is transparent to the programmer. Programmers must be careful to use the XSD macro to translate printf() style format strings that contain natural language. That’s it.

## Throwing and Catching Errors

In the early days of EMS, functions reported failures by returning a special error value. This approach was abandoned because of the code clutter caused by checking nearly every function call. It was replaced by a portable exception handling mechanism using the termination model.

A function signals failure by building an error value, then throwing it to its caller as shown in Figure 6. As a shorthand, we supply the function throw_err_x() which allows the above code to be written in a single line (see Figure 7). The effect of throw_x() or throw_err_x() is to raise an exception: this causes a non-local jump down through the call stack to the nearest exception handler. If no exception handlers are active, then the error is printed to stderr, and the program exits with status 255. As a result of this default behaviour, simple utility programs that exit with an error message when any error occurs are very easy to write.

Exceptions may be caught using the control structure shown in Figure 8. The “THEN_TRY ...’’ clause may be repeated zero or more times. The identifier argument to CATCH is used to declare a variable of type Error_t* which is local to the exception handler. A TRY .. END_TRY block is syntactically a compound statement, and may be written anywhere a statement is legal in C. They may even be nested.

A TRY .. END_TRY block is evaluated in the following manner. First, each body is executed in sequence. If an exception is raised during the execution of a body, then control is immediately transferred to the beginning of the next body. After all of the bodies have been executed, one of two things happens. If an exception occurred during the execution of any body, then the handler is executed with the Error_t* variable pointing to the first error that occurred. If none of the bodies raised an exception, then the handler is skipped.

For example, the following code:

```
TRY
        foo_x()
CATCH (err)
        err_print_x(err, stderr);
END_TRY
```

calls foo_x(); if it raises an exception, then the exception is caught, and the error is printed to stderr.

If you use break, continue, goto or return to break out of the body of a TRY or THEN_TRY clause, then you must call TPOP() before

```
Error_t err;
err_set(&err, sys_errs, (long)ENOMEM, "Couldn't allocate %d bytes", size);
throw_x(&err);
```

**Figure 6**: Creating an error value and throwing it to its caller

```
throw_err_x(sys_errs, (long)ENOMEM, "Couldn't allocate %d bytes", size);
```

**Figure 7**: Combined set and throw functions

```
TRY
        .. body1: statements that may raise an exception ..
THEN_TRY
        .. body2: more statements that may raise an exception ..
CATCH (identifier)
        .. exception handler ..
END_TRY
```

**Figure 8**: Exception catching

transferring control. Otherwise, the stack maintained by the exception handling system will be damaged. TPOP() can only be used to break out of a single TRY statement; it cannot be used to break out of several nested TRY statements at once. For example:

```
TRY
    if (bar_x() == 0) {
        TPOP();
        return;
    }
CATCH (err)
    err_print_x(err, stderr);
END_TRY
```

The exception handling system described in this section is written in portable C, with the help of setjmp(), longjmp() and the C preprocessor. Although its genesis is independent, the implementation is similar to that used by Roberts [Roberts 89]. The need for the TPOP() macro is regrettable, and is an occasional source of bugs. A purpose built preprocessor could eliminate the need for TPOP(), and could also generate slightly better quality C code for the TRY statement.

### Cleaning Up After Errors

It is the responsibility of every library function to clean up properly after detecting an error. There must be no memory leaks, file descriptor leaks, or data structures left in an invalid state after an early error exit.

Consider this function (note that mem_alloc_x() is a wrapper for malloc() that raises exceptions, and mem_free() is a wrapper for free()):

```
void
foo_x()
{
    Foo_t *f1, *f2;

    f1 = mem_alloc_x(sizeof(Foo_t));
    f2 = mem_alloc_x(sizeof(Foo_t));
    ... body of foo_x ...
    mem_free(f1);
    mem_free(f2);
}
```

If the first call to mem_alloc_x() fails, then foo_x will be immediately terminated by an exception. This is the desired effect, and no error handling code is required within foo_x to make this happen. However, if the second call to mem_alloc_x() fails, then there will be a storage leak, because foo_x will exit without freeing f1. Similarly, if the body of foo_x (as represented by the ellipsis) is capable of failing, then neither f1 and f2 will be freed. Our solution to this problem is to use the coding style in Figure 9.

The TRY clause contains resource allocation and the body of foo_x. The THEN_TRY clause frees resources; it is executed whether or not the body fails. The reason that we initialize f1 and f2 to NULL is so that the calls to mem_free() in the THEN_TRY clause will work even if an exception is raised in one of the calls to mem_alloc_x(). mem_free() is guaranteed to ignore a NULL argument, unlike free() on some UNIX systems.

```
void
foo_x()
{
    Foo_t *f1, *f2;

    f1 = NULL;
    f2 = NULL;
    TRY
      f1 = mem_alloc_x(sizeof(Foo_t));
      f2 = mem_alloc_x(sizeof(Foo_t));
      ... body of foo_x ...
    THEN_TRY
      mem_free(f1);
      mem_free(f2);
    CATCH (err)
      throw_x(err);
    END_TRY
}
```

**Figure 9**: Cleaning up after an error

This kind of analysis (for resource leaks) has to be performed every time a library function is written. In order to make the analysis and coding easier to perform, we strictly enforce two conventions. First, all library functions that are capable of raising exceptions have names suffixed by _x. Second, all deallocation routines are required to ignore a NULL argument. The _x convention has proven to be quite useful, because it is otherwise very difficult to tell whether or not a particular stretch of code is capable of raising exceptions, and this is critical for resource leak analysis. It is not convenient to assume that every function call could raise an exception, because we make heavy use of access macros to replace direct access to structure members, and these usually don't raise exceptions.

### Documenting Exceptions

One of the rules that we have tried to enforce is that the error interface provided by each function must be fully documented. After all, this error interface is part of the contract that the function makes with its callers, just as surely as the result and argument types are.

We have run into several problems trying to achieve this goal.

The first problem arises from the fact that it is difficult to document the error interface of high-level functions if the low-level functions that they call don't have well-defined error interfaces. Unfortunately, we have this problem with the UNIX system calls. On many UNIX systems, the set of errors generated by each system call (and the semantics of system calls when an error is detected) is only partially documented. Furthermore, the error interface for system calls is not portable: it changes from one system to the next. Consequently, EMS library functions that perform I/O currently have a poorly documented, system dependent error interface, and writing error handlers for code that does I/O sometimes involves experimentation to find out the names of the error numbers of interest, combined with #ifdefs to check for different error numbers on different machines. The obvious solution to this problem, which we haven't had time to pursue, is to define a portable error interface for each system call, then to work out the mapping from system error ids to portable error ids for each system call and machine type.

The second problem arises from the fact that the error interface for a function tends to be defined as the union of the error interfaces for all of the functions that it calls. Not only does this lead to large, cluttered error interfaces, but it also means that error interfaces tend to change over time as a result of maintenance, and thus the documentation for error interfaces tends to drift out of sync with reality. This can be viewed as a documentation and maintenance problem, in which case the solution might be to attempt to generate the documentation for error interfaces automatically, using a code analysis tool. However, this problem can also be viewed as a design problem: in some cases, programmers are not making the effort to design a high-level, abstract error interface that matches the abstraction provided by the function; instead, they are letting the error interface default to whatever their code does.

Many strongly typed languages with built-in exception mechanisms either permit or require you to declare the error interface of a function as part of its type. (C++ and Modula-3 are examples.) This declaration consists of a fixed list of error ids. In our experience, life is not that simple. If you are doing object oriented programming (as we do), then what you will find is that different subclasses of a base class A will implement different error interfaces for the virtual function inherited from A. For example, we have a class called Stream (similar to a stdio FILE). The set of errors that can occur in (eg) the write_x virtual function depends on which subclass of Stream you are writing to. As a result, the error interface for a function that takes a Stream as an argument depends on which subclass of Stream is actually passed in. Language designers might wish to consider polymorphic exception sets within function signatures to deal with this issue.

## Testing

An important part of our error handling package (and a topic rarely discussed in the literature on exception handling) is testing. Full adherence to the error handling discipline described here adds a noticeable amount of complexity to our code, mostly in the form of exception handlers within library functions which deallocate resources and restore data structure invariants before forwarding an exception. These exception handlers are rarely executed in production code, and are therefore a fertile breeding ground for bugs. Fortunately, we have developed a simple, yet powerful mechanism for testing these exception handlers. This mechanism works by triggering fake exceptions in low-level functions under the control of command line arguments. These fake exceptions set off a cascade of intermediate exception handlers, thereby exercising them. When used in conjunction with a library regression test program, this mechanism can be made to exercise all of the exception handling code in a library.

A control point for triggering a fake exception is placed by calling the macro xtrap_x(), which takes an error id as an argument. These control points are placed in low level library functions, at points where an exception could potentially be raised by natural means. Only a handful of calls to xtrap_x are needed in EMS; the single call to xtrap_x in mem_alloc_x (our wrapper for malloc()) is sufficient to test 75-80% of our exception handling code.

EMS has a mechanism for passing debug options into a program for use by library routines. These debug options can either be placed in an environment variable, or they can be supplied as arguments to the command line flag -V, which all EMS programs support. The xtrap mechanism is triggered using the command line argument -V xt=i, which causes the i-th dynamic invocation of xtrap_x to raise an exception. The command line argument -V xc causes a program to run to completion, then print out the number of times the xtrap_x macro was executed. To exercise a library, we write a test program which exercises all of the functions in the library. Then we run the program with -V xc, which gives us the number n. Finally, we run the program n times (from a shell script), supplying the i-th invocation with the argument -V xt=i. This is usually sufficient to exercise 99% of the exception handling code in the library (assuming that the test program is written to provide full coverage of the library). When this testing technique is combined with the debugging version of our storage allocator (which detects bad calls to free and storage leaks), we can detect most problems involving improper releasing of resources in exception handlers.

### Evaluation

The EMS approach to error handling is a success. Our code is robust, well behaved, and testable, and programmer acceptance of the approach is high. There is an initial cost in learning how to use the error handling system, especially for junior programmers, who can be a little uncomfortable with the fact that most functions they call are liable to perform a non-local jump upon encountering an error. However, once programmers get past the initial learning hump, they seem to like the system, since using it leads to cleaner looking code that is less obscured by error handling than equivalent code which checks each function call for error return values.

Under the old regime of signalling errors by return codes, the most common bug is failing to check the return code. This can lead to nasty behaviour, like core dumps. Under the new regime of using exceptions to signal errors, the most common bug is failing to catch exceptions for the purpose of freeing resources, and this leads to resource leaks[1]. The new style of bug has fewer harmful effects on the overall robustness of the code.

There is still room for improvement. A more flexible method for organizing error ids into a hierarchy, so that programmers can test an error for membership in a group, and a way of associating integer and string parameters with an error, would be welcome improvements. So would a lint-like tool for detecting common bugs in exception-handling code, and a tool to automate the documentation of error interfaces by scanning source code.

The EMS error handling system compares quite favourably to other C exception handling packages. Its greatest strength is the Error_t structure, which incorporates the novel idea of a stack of error interpretations, and provides standard ways to print arbitrary error descriptions, and to transmit them across an IPC connection. Roberts' system [Roberts 89] provides a mechanism for raising and handling exceptions very similar to the EMS system, but errors are represented by a pair consisting of an error id and an integer parameter. Allman's system [Allman 85] provides a flexible way to organize error ids into groups, has character string parameters which are accessible to error handlers, and integrates UNIX signals with the exception system. On the minus side, his system requires assembly language support, and has a much less convenient syntax for exception handlers. Allman also supports the more general resumption model of exception handling. My feeling is that terminate and resume style exceptions should be signalled and handled by different mechanisms; see the appendix.

### A Plea For Standardization

The benefits of an exception handling system are strongest when it is used everywhere. Accordingly, the EMS utilities library contains exception-raising replacements or wrappers for many of the most commonly used C library functions. We even went so far as to implement a complete replacement for stdio with better error handling[2] (and of course better performance). Unfortunately, we don't get the full benefit of our error handling system when we use libraries, such as those for the X Window System, that were written by other people. Because the C error system (errno, perror, etc.) is not extensible, every library implementor is forced to create their own error handling system, and application programmers are stuck with the job of knitting these different error handling systems together.

We think the world would be a better place if the C community had a standard, extensible, and well designed system for fault and failure handling which all library implementors used. This probably won't happen in the C community, but there is still hope for new languages such as C++. Implementations of C++ that support exception handling are just now starting to appear. Unfortunately, syntax for raising and handling exceptions is not enough: there should also be standard conventions for using it, a standard way to print errors, and exceptions raised in all standard library routines upon failure.

### References

[Allman 85] Eric Allman and David Been. "An Exception Handler for C, " Proceedings of the Summer 1985 USENIX Conference. Portland, Oregon, 1985.

[Darwin 85] I. Darwin and G. Collyer. "Can't happen -or- /* NOTREACHED */ -or- Real Programs Dump Core," Proceedings of the Winter 1985 USENIX Conference. Dallas, Texas, January 1985.

[Ellis 90] Margaret A. Ellis and Bjarne Stroustrup. The Annotated C++ Reference Manual. Addison Wesley, 1990.

[Goodenough 75] John B. Goodenough. "Exception Handling: issues and a proposed notation," Communications of the ACM. Vol. 18, no. 12, December 1975.

[Liskov 79] Barbara A. Liskov and Alan Snyder. "Exception Handling in CLU," IEEE Transactions on Software Engineering. Vol. SE-5, no.

[1] Using the debug version of the EMS memory allocator, resource leaks are not difficult to diagnose, since we provide a facility for listing the memory blocks which are still allocated at program exit time; this list gives the file name and line number of the call to mem_alloc_x for each allocated block.

[2] Stdio doesn't tell you what went wrong when an error occurs. We discovered the hard way that the contents of errno are not to be trusted after a stdio error.

6, November 1979.

[Meyer 88] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall, 1988.

[Nelson 91] Greg Nelson. Systems Programming with Modula-3. Prentice Hall, 1991.

[Randell 75] Brian Randell. "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering. Vol. SE-1, no. 2, June 1975.

[Roberts 89] Eric S. Roberts. "Implementing Exceptions in C," Research Report 40, Digital Systems Research Center, March 21, 1989.

[Steele 90] Guy L. Steele Jr. Common Lisp. Digital Press, 1990.

[Yemeni 85] Shaula Yemeni and Daniel Berry. "A modular verifiable exception-handling mechanism," Transactions on Programming Languages and Systems. Vol. 7, no. 2, April 1985.

## Author Information

After 7 years of hacking UNIX, C, Macintosh, and graphical user interfaces, Doug Moen received his B.I.S. from the University of Waterloo in 1987. His bachelor's thesis was the design of a programming language (Goal) with powerful abstraction mechanisms and a polymorphic type system based on dependent types. In 1988, after graduating, Doug joined Interactive Image Technologies where he made major contributions to an object-oriented, platform independent graphical user interface library, and was the principle designer and architect for HyperCase, a multi-media authoring tool. In 1989, Doug moved to Nixdorf, where he has made substantial contributions to the design and implementation of EMS, a programmers toolkit for building document image processing systems. Doug is now available for employment. Reach him at doug@sni.ca, or at (416) 977-4907, or at 77 Carlton Street #1504, Toronto, Ontario, M5B 2J7.

## Appendix: Notify and Signal Conditions

This paper has dealt with 2 kinds of exceptional conditions which can be detected by a library function: faults and failures. Both of these kinds of conditions result in the termination of a function call once they are detected. But there is a third class of exceptional conditions: these are conditions which need not prevent the function from completing its task, but which may nevertheless need to be reported to the caller before the function has completed its task. Goodenough [Goodenough 75] distinguishes two types of exceptional conditions which fall into this category, which he calls *notify* and *signal* conditions. A handler for a *notify* condition is prohibited from terminating the operation. A handler for a *signal* condition is given the choice of terminating the operation, or fixing the problem and resuming it.

For an example of a notify condition, consider pclose(3), which repeatedly calls wait(2) until the process associated with the argument stream has exited. The exit statuses of child processes which are different from the process that pclose() is attempting to wait for are simply thrown away, and programs that call pclose() have no way of obtaining this information. This is a design flaw in pclose() which could be rectified through the use of a mechanism for reporting notify conditions.

For an example of a signal condition, consider a function which is communicating with a remote process over a network connection. If the remote process has apparently stopped talking, then the function has two choices: it can report a failure, or it can keep trying to communicate with its peer, on the assumption that the peer might resume communication in a few seconds. In practice, timeouts are often used in this kind of situation. A better approach might be to report a signal condition, and let the caller attempt to fix the problem, or obtain advice from the user on whether to retry or abort.

A second example of a signal condition is a memory allocator which gives the caller an opportunity to free cached memory blocks before reporting a failure.

As I have tried to show, signal and notify conditions are real. The question is, what combination of language features and programming conventions are most appropriate for dealing with them?

One approach is to incorporate the reporting of signal and notify conditions into the same exception handling mechanism that is used to report failures. This leads to the retry model of exception handling. In this model, when an exception is raised, the context raising the exception is not immediately terminated. Instead, the exception handler is given the choice of terminating the function that raised the exception, or resuming it. The Mesa programming language supported this model of exception handling; so does Common Lisp [Steele 90]; and so does Allman and Been's exception handler for C [Allman 85].

An evident disadvantage of implementing the retry model of exception handling in C is that it is necessary to make exception handlers into separate functions. Not only is this grossly inconvenient, but the exception handlers do not have access to the local variables associated with the statement block that they are associated with.

The retry model of exception handling provides dynamically scoped handlers for signal and notify conditions that are associated with specific blocks of code. It is not clear that this is the best scoping regimen. Alternatives are to associate exception handlers with modules, or to associate them with objects. Handlers with module scope can be implemented by registering a callback function with the appropriate module. Handlers with object scope can be implemented using object oriented programming, by overriding a virtual function in the class of the object.

I have not yet developed a personal philosophy concerning the proper treatment of signal and notify conditions. However, I would like to show how dynamically scoped handlers for these conditions could be added to the EMS error handling system. The technique could probably be adopted to work with any C-based termination-model exception handling system.

The function

```
int notify(Error_t *)
```

is used by a function to notify its client of an abnormal condition; it is used to implement both ''signal'' and ''notify'' exceptions as described above. The Error_t structure is used to describe the abnormal condition. The value returned by notify() is one of the values:

N_ABORT The client wishes the function to abort by raising an exception.

N_RETRY The client wishes the function to keep trying.

N_IGNORE The client ignored the notification.

A client can catch a notification using the following control structure:

```
NOTIFY (fun)
   ... code which may raise
       a notification ...
END_NOTIFY
```

The argument to NOTIFY is a function pointer with type

```
int (*fun) (Error_t *)
```

This function should examine the Error structure (and in particular, the error id), and return N_ABORT, N_RETRY or N_IGNORE. The return value N_IGNORE means that the notification handler function did not recognize this particular notification.

NOTIFY ... END_NOTIFY blocks can be nested in the same way that TRY ... END_TRY blocks can. When a notification is raised, notify() searches the stack of NOTIFY blocks, calling each notification handler function in turn until one of them returns a value different from N_IGNORE. If no handlers are present, or no handler is willing to handle the condition, then notify() returns N_IGNORE.