A Common Error Description Library for UNIX

Ken Raeburn
Bill Sommerfeld

MIT Student Information Processing Board

last updated 1 January 1989
for version 1.2
DRAFT COPY ONLY

Abstract

UNIX has always had a clean and simple system call interface, with a standard set of error codes passed between the kernel and user programs. Unfortunately, the same cannot be said of many of the libraries layered on top of the primitives provided by the kernel. Typically, each one has used a different style of indicating errors to their callers, leading to a total hodgepodge of error handling, and considerable amounts of work for the programmer. This paper describes a library and associated utilities which allows a more uniform way for libraries to return errors to their callers, and for programs to describe errors and exceptional conditions to their users.

0.1 Why com_err?

In building application software packages, a programmer often has to deal with a number of libraries, each of which can use a different error-reporting mechanism. Sometimes one of two values is returned, indicating simply SUCCESS or FAILURE, with no description of errors encountered. Sometimes it is an index into a table of text strings, where the name of the table used is dependent on the library being used when the error is generated; since each table starts numbering at 0 or 1, additional information as to the source of the error code is needed to determine which table to look at. Sometimes no text messages are supplied at all, and the programmer must supply them at any point at which he may wish to report error conditions. Often, a global variable is assigned some value describing the error, but the programmer has to know in each case whether to look at `errno`, `h_errno`, the return value from `hes_err()`, or whatever other variables or routines are specified. And what happens if something in the procedure of examining or reporting the error changes the same variable?

The package we have developed is an attempt to present a common error-handling mechanism to manipulate the most common form of error code in a fashion that does not have the problems listed above.

A list of up to 256 text messages is supplied to a translator we have written, along with the three- to four-character "name" of the error table. The library using this error table need only call a routine generated from this error-table source to make the table "known" to the com_err library, and any error code the library generates can be converted to the corresponding error message. There is also a default format for error codes accidentally returned before making the table known, which is of the form '`unknown code foo 32`', where '`foo`' would be the name of the table.

0.2 Error codes

Error codes themselves are 32 bit (signed) integers, of which the high order 24 bits are an identifier of which error table the error code is from, and the low order 8 bits are a sequential error number within the table. An error code may thus be easily decomposed into its component parts. Only the lowest 32 bits of an error code are considered significant on systems which support wider values.

Error table 0 is defined to match the UNIX system call error table (`sys_errlist`); this allows `errno` values to be used directly in the library (assuming that `errno` is of a type with the same width as `long`). Other error table numbers are formed by compacting together the first four characters of the error table name. The mapping between characters in the name and numeric values in the error code are defined in a system-independent fashion, so that two systems that can pass integral values between them can reliably pass error codes without loss of meaning; this should work even if the character sets used are not the same. (However, if this is to be done, error table 0 should be avoided, since the local system call error tables may differ.)

Any variable which is to contain an error code should be declared `long`. The draft proposed American National Standard for C (as of May, 1988) requires that `long` variables be at least 32 bits; any system which does not support 32-bit `long` values cannot make use of this package (nor much other software that assumes an ANSI-C environment base) without significant effort.

0.3 Error table source file

The error table source file begins with the declaration of the table name, as

`error_table` *tablename*

Individual error codes are specified with

`error_code` *ERROR_NAME*`, "`*text message*`"`

where '`ec`' can also be used as a short form of '`error_code`'. To indicate the end of the table, use '`end`'. Thus, a (short) sample error table might be:

```
error_table     dsc

error_code      DSC_DUP_MTG_NAME,
                "Meeting already exists"

ec              DSC_BAD_PATH,
                "A bad meeting pathname was given"

ec              DSC_BAD_MODES,
                "Invalid mode for this access control list"

end
```

0.4 The error-table compiler

The error table compiler is named `compile_et`. It takes one argument, the pathname of a file (ending in '`.et`', e.g., '`dsc_err.et`') containing an error table source file. It parses the error table, and generates two output files – a C header file ('`discuss_err.h`') which contains definitions of the numerical values of the error codes defined in the error table, and a C source file which should be compiled and linked with the executable. The header file must be included in the source of a module which wishes to reference the error codes defined; the object module generated from the C code may be linked in to a program which wishes to use the printed forms of the error codes.

This translator accepts a `-language` *lang* argument, which determines for which language (or language variant) the output should be written. At the moment, *lang* is currently limited to *`ANSI-C`* and *`K&R-C`*, and some abbreviated forms of each. Eventually, this will be extended to include some support for C++. The default is currently *`K&R-C`*, though the generated sources will have ANSI-C code conditionalized on the symbol `__STDC__`.

0.5 Run-time support routines

Any source file which uses the routines supplied with or produced by the com_err package should include the header file '`<com_err.h>`'. It contains declarations and definitions which may be needed on some systems. (Some functions cannot be referenced properly without the return type declarations in this file. Some functions may work properly on most architectures even without the header file, but relying on this is not recommended.)

The run-time support routines and variables provided via this package include the following:

```
void initialize_xxxx_error_table (void);
```

One of these routines is built by the error compiler for each error table. It makes the *xxxx* error table "known" to the error reporting system. By convention, this routine should be called in the initialization routine of the *xxxx* library. If the library has no initialization routine, some combination of routines which form the core of the library should ensure that this routine is called. It is not advised to leave it the caller to make this call.

There is no harm in calling this routine more than once.

```
#define ERROR_TABLE_BASE_xxxx nnnnnL
```

This symbol contains the value of the first error code entry in the specified table. This rarely needs be used by the programmer.

```
const char *error_message (long code);
```

This routine returns the character string error message associated with `code`; if this is associated with an unknown error table, or if the code is associated with a known error table but the code is not in the table, a string of the form '`Unknown code` *xxxx nn*' is returned, where *xxxx* is the error table name produced by reversing the compaction performed on the error table number implied by that error code, and *nn* is the offset from that base value.

Although this routine is available for use when needed, its use should be left to circumstances which render `com_err` (below) unusable.

```
void com_err (const char *whoami,  /* module reporting error */
              long code,           /* error code */
              const char *format,  /* format for additional detail */
              ...);                /*  (extra parameters) */
```

This routine provides an alternate way to print error messages to standard error; it allows the error message to be passed in as a parameter, rather than in an external variable. *Provide grammatical context for "message."*

If *format* is `(char *)NULL`, the formatted message will not be printed. *format* may not be omitted.

```
#include <stdarg.h>

void com_err_va (const char *whoami,
                 long code,
                 const char *format,
                 va_list args);
```

This routine provides an interface, equivalent to `com_err` above, which may be used by higher-level variadic functions (functions which accept variable numbers of arguments).

```
#include <stdarg.h>

void (*set_com_err_hook (void (*proc) ())) ();

void (*proc) (const char *whoami, long code, va_list args);

void reset_com_err_hook ();
```

These two routines allow a routine to be dynamically substituted for '`com_err`'. After '`set_com_err_hook`' has been called, calls to '`com_err`' will turn into calls to the new hook routine. '`reset_com_err_hook`' turns off this hook. This may intended to be used in daemons (to use a routine which calls *syslog(3)*), or in a window system application (which could pop up a dialogue box).

If a program is to be used in an environment in which simply printing messages to the `stderr` stream would be inappropriate (such as in a daemon program which runs without a terminal attached), `set_com_err_hook` may be used to redirect output from `com_err`. The following is an example of an error handler which uses *syslog(3)* as supplied in BSD 4.3:

```
#include <stdio.h>
#include <stdarg.h>
#include <syslog.h>

/* extern openlog (const char * name, int logopt, int facility); */
/* extern syslog (int priority, char * message, ...); */

void hook (const char * whoami, long code,
           const char * format, va_list args)
{
    char buffer[BUFSIZ];
    static int initialized = 0;
    if (!initialized) {
        openlog (whoami,
                 LOG_NOWAIT|LOG_CONS|LOG_PID|LOG_NDELAY,
                 LOG_DAEMON);
        initialized = 1;
    }
    vsprintf (buffer, format, args);
    syslog (LOG_ERR, "%s %s", error_message (code), buffer);
}
```

After making the call `set_com_err_hook (hook);`, any calls to `com_err` will result in messages being sent to the *syslogd* daemon for logging. The name of the program, '`whoami`', is supplied to the '`openlog()`' call, and the message is formatted into a buffer and passed to `syslog`.

Note that since the extra arguments to `com_err` are passed by reference via the `va_list` value `args`, the hook routine may place any form of interpretation on them, including ignoring them. For consistency, `printf`-style interpretation is suggested, via `vsprintf` (or `_doprnt` on BSD systems without full support for the ANSI C library).

0.6 Coding Conventions

The following conventions are just some general stylistic conventions to follow when writing robust libraries and programs. Conventions similar to this are generally followed inside the UNIX kernel and most routines in the Multics operating system. In general, a routine either succeeds (returning a zero error code, and doing some side effects in the process), or it fails, doing minimal side effects; in any event, any invariant which the library assumes must be maintained.

In general, it is not in the domain of non user-interface library routines to write error messages to the user's terminal, or halt the process. Such forms of "error handling" should be reserved for failures of internal invariants and consistancy checks only, as it provides the user of the library no way to clean up for himself in the event of total failure.

Library routines which can fail should be set up to return an error code. This should usually be done as the return value of the function; if this is not acceptable, the routine should return a "null" value, and put the error code into a parameter passed by reference.

Routines which use the first style of interface can be used from user-interface levels of a program as follows:

```
{
    if ((code = initialize_world(getuid(), random())) != 0) {
        com_err("demo", code,
                "when trying to initialize world");
        exit(1);
    }
    if ((database = open_database("my_secrets", &code))==NULL) {
        com_err("demo", code,
                "while opening my_secrets");
        exit(1);
    }
}
```

A caller which fails to check the return status is in error. It is possible to look for code which ignores error returns by using lint; look for error messages of the form "foobar returns value which is sometimes ignored" or "foobar returns value which is always ignored."

Since libraries may be built out of other libraries, it is often necessary for the success of one routine to depend on another. When a lower level routine returns an error code, the middle level routine has a few possible options. It can simply return the error code to its caller after doing some form of cleanup, it can substitute one of its own, or it can take corrective action of its own and continue normally. For instance, a library routine which makes a "connect" system call to make a network connection may reflect the system error code `ECONNREFUSED` (Connection refused) to its caller, or it may return a "server not available, try again later," or it may try a different server.

Cleanup which is typically necessary may include, but not be limited to, freeing allocated memory which will not be needed any more, unlocking concurrancy locks, dropping reference counts, closing file descriptors, or otherwise undoing anything which the procedure did up to this point. When there are a lot of things which can go wrong, it is generally good to write one block of error-handling code which is branched to, using a goto, in the event of failure. A common source of errors in UNIX programs is failing to close file descriptors on error returns; this leaves a number of "zombied" file descriptors open, which eventually causes the process to run out of file descriptors and fall over.

```
{
    FILE *f1=NULL, *f2=NULL, *f3=NULL;
    int status = 0;

    if ( (f1 = fopen(FILE1, "r")) == NULL) {
        status = errno;
```

```
        goto error;
    }

    /*
     * Crunch for a while
     */

    if ( (f2 = fopen(FILE2, "w")) == NULL) {
        status = errno;
        goto error;
    }

    if ( (f3 = fopen(FILE3, "a+")) == NULL) {
        status = errno;
            goto error;
    }

    /*
     * Do more processing.
     */
    fclose(f1);
    fclose(f2);
    fclose(f3);
    return 0;

error:
    if (f1) fclose(f1);
    if (f2) fclose(f2);
    if (f3) fclose(f3);
    return status;
}
```

0.7 Building and Installation

The distribution of this package will probably be done as a compressed "tar"-format file available via anonymous FTP from SIPB.MIT.EDU. Retrieve '`pub/com_err.tar.Z`' and extract the contents. A subdirectory `profiled` should be created to hold objects compiled for profiling. Running "make all" should then be sufficient to build the library and error-table compiler. The files '`libcom_err.a`', '`libcom_err_p.a`', '`com_err.h`', and '`compile_et`' should be installed for use; '`com_err.3`' and '`compile_et.1`' can also be installed as manual pages.

Potential problems:

- Use of `strcasecmp`, a routine provided in BSD for case-insensitive string comparisons. If an equivalent routine is available, you can modify `CFLAGS` in the makefile to define `strcasecmp` to the name of that routine.
- Compilers that defined `__STDC__` without providing the header file `<stdarg.h>`. One such example is Metaware's High "C" compiler, as provided at Project Athena on the

IBM RT/PC workstation; if `__HIGHC__` is defined, it is assumed that `<stdarg.h>` is not available, and therefore `<varargs.h>` must be used. If the symbol `VARARGS` is defined (e.g., in the makefile), `<varargs.h>` will be used.

- If your linker rejects symbols that are simultaneously defined in two library files, edit '`Makefile`' to remove '`perror.c`' from the library. This file contains a version of *perror(3)* which calls `com_err` instead of calling `write` directly.

As I do not have access to non-BSD systems, there are probably bugs present that may interfere with building or using this package on other systems. If they are reported to me, they can probably be fixed for the next version.

0.8 Bug Reports

Please send any comments or bug reports to the principal author: Ken Raeburn, `Raeburn@Athena.MIT.EDU`.

0.9 Acknowledgements

I would like to thank: Bill Sommerfeld, for his help with some of this documentation, and catching some of the bugs the first time around; Honeywell Information Systems, for not killing off the *Multics* operating system before I had an opportunity to use it; Honeywell's customers, who persuaded them not to do so, for a while; Ted Anderson of CMU, for catching some problems before version 1.2 left the nest; Stan Zanarotti and several others of MIT's Student Information Processing Board, for getting us started with "discuss," for which this package was originally written; and everyone I've talked into — I mean, asked to read this document and the "man" pages.