Error Reporting

Many functions in the GNU C library detect and report error conditions, and sometimes your programs need to check for these error conditions. For example, when you open an input file, you should verify that the file was actually opened correctly, and print an error message or take other appropriate action if the call to the library function failed.

This chapter describes how the error reporting facility works. Your program should include the header file `errno.h' to use this facility.

Checking for Errors

Most library functions return a special value to indicate that they have failed. The special value is typically -1, a null pointer, or a constant such as EOF that is defined for that purpose. But this return value tells you only that an error has occurred. To find out what kind of error it was, you need to look at the error code stored in the variable errno. This variable is declared in the header file `errno.h'.

Variable: volatile int **errno**

The variable errno contains the system error number. You can change the value of errno.

Since errno is declared volatile, it might be changed asynchronously by a signal handler; see section Defining Signal Handlers. However, a properly written signal handler saves and restores the value of errno, so you generally do not need to worry about this possibility except when writing signal handlers.

The initial value of errno at program startup is zero. Many library functions are guaranteed to set it to certain nonzero values when they encounter certain kinds of errors. These error conditions are listed for each function. These functions do not change errno when they succeed; thus, the value of errno after a successful call is not necessarily zero, and you should not use errno to determine *whether* a call failed. The proper way to do that is documented for each function. *If* the call failed, you can examine errno.

Many library functions can set errno to a nonzero value as a result of calling other library functions which might fail. You should assume that any library function might alter errno when the function returns an error.

> **Portability Note:** ISO C specifies `errno` as a "modifiable lvalue" rather than as a variable, permitting it to be implemented as a macro. For example, its expansion might involve a function call, like `*__errno ()`. In fact, that is what it is on the GNU system itself. The GNU library, on non-GNU systems, does whatever is right for the particular system.
>
> There are a few library functions, like `sqrt` and `atan`, that return a perfectly legitimate value in case of an error, but also set `errno`. For these functions, if you want to check to see whether an error occurred, the recommended method is to set `errno` to zero before calling the function, and then check its value afterward.

All the error codes have symbolic names; they are macros defined in `errno.h`. The names start with `'E'` and an upper-case letter or digit; you should consider names of this form to be reserved names. See section Reserved Names.

The error code values are all positive integers and are all distinct, with one exception: `EWOULDBLOCK` and `EAGAIN` are the same. Since the values are distinct, you can use them as labels in a `switch` statement; just don't use both `EWOULDBLOCK` and `EAGAIN`. Your program should not make any other assumptions about the specific values of these symbolic constants.

The value of `errno` doesn't necessarily have to correspond to any of these macros, since some library functions might return other error codes of their own for other situations. The only values that are guaranteed to be meaningful for a particular library function are the ones that this manual lists for that function.

On non-GNU systems, almost any system call can return `EFAULT` if it is given an invalid pointer as an argument. Since this could only happen as a result of a bug in your program, and since it will not happen on the GNU system, we have saved space by not mentioning `EFAULT` in the descriptions of individual functions.

In some Unix systems, many system calls can also return `EFAULT` if given as an argument a pointer into the stack, and the kernel for some obscure reason fails in its attempt to extend the stack. If this ever happens, you should probably try using statically or dynamically allocated memory instead of stack memory on that system.

Error Messages

The library has functions and variables designed to make it easy for your program to report informative error messages in the customary format about the failure of a library call. The functions `strerror` and `perror` give you the standard error message for a given error code; the variable `program_invocation_short_name` gives you convenient access to the name of the program that encountered the error.

Function: char * **strerror** *(int errnum)*

The `strerror` function maps the error code (see section Checking for Errors) specified by the *errnum* argument to a descriptive error message string. The return value is a pointer to this string.

The value *errnum* normally comes from the variable `errno`.

You should not modify the string returned by `strerror`. Also, if you make subsequent calls to `strerror`, the string might be overwritten. (But it's guaranteed that no library function ever calls `strerror` behind your back.)

The function `strerror` is declared in `` `string.h' ``.

Function: char * **strerror_r** *(int errnum, char *buf, size_t n)*

The `strerror_r` function works like `strerror` but instead of returning the error message in a statically allocated buffer shared by all threads in the process, it returns a private copy for the thread. This might be either some permanent global data or a message string in the user supplied buffer starting at *buf* with the length of *n* bytes.

At most *n* characters are written (including the NUL byte) so it is up to the user to select the buffer large enough.

This function should always be used in multi-threaded programs since there is no way to guarantee the string returned by `strerror` really belongs to the last call of the current thread.

This function `strerror_r` is a GNU extension and it is declared in `` `string.h' ``.

Function: void **perror** *(const char *message)*

This function prints an error message to the stream `stderr`; see section Standard Streams.

If you call `perror` with a *message* that is either a null pointer or an empty string, `perror` just prints the error message corresponding to `errno`, adding a trailing newline.

If you supply a non-null *message* argument, then `perror` prefixes its output with this string. It adds a colon and a space character to separate the *message* from the error string corresponding to `errno`.

The function `perror` is declared in `` `stdio.h' ``.

`strerror` and `perror` produce the exact same message for any given error code; the precise text varies from system to system. On the GNU system, the messages are fairly short; there are no multi-line messages or embedded newlines. Each error message begins with a capital letter and does not include any terminating punctuation.

Compatibility Note: The `strerror` function is a new feature of ISO C. Many older C systems do not support this function yet.

Many programs that don't read input from the terminal are designed to exit if any system call fails. By convention, the error message from such a program should start with the program's name, sans directories. You can find that name in the variable `program_invocation_short_name`; the full file name is stored the variable`program_invocation_name`.

Variable: char * **program_invocation_name**

This variable's value is the name that was used to invoke the program running in the current process. It is the same as `argv[0]`. Note that this is not necessarily a useful file name; often it contains no directory names. See section Program Arguments.

Variable: char * **program_invocation_short_name**

This variable's value is the name that was used to invoke the program running in the current process, with directory names removed. (That is to say, it is the same as `program_invocation_name` minus everything up to the last slash, if any.)

The library initialization code sets up both of these variables before calling `main`.

Portability Note: These two variables are GNU extensions. If you want your program to work with non-GNU libraries, you must save the value of `argv[0]` in`main`, and then strip off the directory names yourself. We added these extensions to make it possible to write self-contained error-reporting subroutines that require no explicit cooperation from `main`.

Here is an example showing how to handle failure to open a file correctly. The function `open_sesame` tries to open the named file for reading and returns a stream if successful. The `fopen` library function returns a null pointer if it couldn't open the file for some reason. In that situation, `open_sesame` constructs an appropriate error message using the `strerror` function, and terminates the program. If we were going to make some other library calls before passing the error code to `strerror`, we'd have to save it in a local variable instead, because those other library functions might overwrite `errno` in the meantime.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FILE *
open_sesame (char *name)
{
  FILE *stream;

  errno = 0;
  stream = fopen (name, "r");
  if (stream == NULL)
    { fprintf (stderr, "%s: Couldn't open file %s; %s\n",
      program_invocation_short_name, name, strerror (errno));
      exit (EXIT_FAILURE);
    }
  else
    return stream;
}
```

Error Codes

The error code macros are defined in the header file `errno.h'. All of them expand into integer constant values. Some of these error codes can't occur on the GNU system, but they can occur using the GNU library on other systems.

Macro: int **EPERM**
: Operation not permitted; only the owner of the file (or other resource) or processes with special privileges can perform the operation.

Macro: int **ENOENT**
: No such file or directory. This is a "file doesn't exist" error for ordinary files that are referenced in contexts where they are expected to already exist.

Macro: int **ESRCH**
: No process matches the specified process ID.

Macro: int **EINTR**
: Interrupted function call; an asynchronous signal occurred and prevented completion of the call. When this happens, you should try the call again.

  You can choose to have functions resume after a signal that is handled, rather than failing with `EINTR`; see section Primitives Interrupted by Signals.

Macro: int **EIO**
: Input/output error; usually used for physical read or write errors.

Macro: int **ENXIO**
: No such device or address. The system tried to use the device represented by a file you specified, and it couldn't find the device. This can mean that the device file was installed incorrectly, or that the physical device is missing or not correctly attached to the computer.

Macro: int **E2BIG**
: Argument list too long; used when the arguments passed to a new program being executed with one of the `exec` functions (see section Executing a File) occupy too much memory space. This condition never arises in the GNU system.

Macro: int **ENOEXEC**
: Invalid executable file format. This condition is detected by the `exec` functions; see section Executing a File.

Macro: int **EBADF**
: Bad file descriptor; for example, I/O on a descriptor that has been closed or reading from a descriptor open only for writing (or vice versa).

Macro: int **ECHILD**

There are no child processes. This error happens on operations that are supposed to manipulate child processes, when there aren't any processes to manipulate.

Macro: int **EDEADLK**

Deadlock avoided; allocating a system resource would have resulted in a deadlock situation. The system does not guarantee that it will notice all such situations. This error means you got lucky and the system noticed; it might just hang. See section File Locks, for an example.

Macro: int **ENOMEM**

No memory available. The system cannot allocate more virtual memory because its capacity is full.

Macro: int **EACCES**

Permission denied; the file permissions do not allow the attempted operation.

Macro: int **EFAULT**

Bad address; an invalid pointer was detected. In the GNU system, this error never happens; you get a signal instead.

Macro: int **ENOTBLK**

A file that isn't a block special file was given in a situation that requires one. For example, trying to mount an ordinary file as a file system in Unix gives this error.

Macro: int **EBUSY**

Resource busy; a system resource that can't be shared is already in use. For example, if you try to delete a file that is the root of a currently mounted filesystem, you get this error.

Macro: int **EEXIST**

File exists; an existing file was specified in a context where it only makes sense to specify a new file.

Macro: int **EXDEV**

An attempt to make an improper link across file systems was detected. This happens not only when you use `link` (see section Hard Links) but also when you rename a file with `rename` (see section Renaming Files).

Macro: int **ENODEV**

The wrong type of device was given to a function that expects a particular sort of device.

Macro: int **ENOTDIR**

A file that isn't a directory was specified when a directory is required.

Macro: int **EISDIR**

File is a directory; you cannot open a directory for writing, or create or remove hard links to it.

Macro: int **EINVAL**

Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function.

Macro: int **EMFILE**

The current process has too many files open and can't open any more. Duplicate descriptors do count toward this limit.

In BSD and GNU, the number of open files is controlled by a resource limit that can usually be increased. If you get this error, you might want to increase the`RLIMIT_NOFILE` limit or make it unlimited; see section Limiting Resource Usage.

Macro: int **ENFILE**

There are too many distinct file openings in the entire system. Note that any number of linked channels count as just one file opening; see section Linked Channels. This error never occurs in the GNU system.

Macro: int **ENOTTY**

Inappropriate I/O control operation, such as trying to set terminal modes on an ordinary file.

Macro: int **ETXTBSY**

An attempt to execute a file that is currently open for writing, or write to a file that is currently being executed. Often using a debugger to run a program is considered having it open for writing and will cause this error. (The name stands for "text file busy".) This is not an error in the GNU system; the text is copied as necessary.

Macro: int **EFBIG**

File too big; the size of a file would be larger than allowed by the system.

Macro: int **ENOSPC**

No space left on device; write operation on a file failed because the disk is full.

Macro: int **ESPIPE**

Invalid seek operation (such as on a pipe).

Macro: int **EROFS**

An attempt was made to modify something on a read-only file system.

Macro: int **EMLINK**

Too many links; the link count of a single file would become too large. `rename` can cause this error if the file being renamed already has as many links as it can take (see section Renaming Files).

Macro: int **EPIPE**

Broken pipe; there is no process reading from the other end of a pipe. Every library function that returns this error code also generates a `SIGPIPE` signal; this signal terminates the program if not handled or blocked. Thus, your program will never actually see `EPIPE` unless it has handled or blocked `SIGPIPE`.

Macro: int **EDOM**

Domain error; used by mathematical functions when an argument value does not fall into the domain over which the function is defined.

Macro: int **ERANGE**

Range error; used by mathematical functions when the result value is not representable because of overflow or underflow.

Macro: int **EAGAIN**

Resource temporarily unavailable; the call might work if you try again later. The macro `EWOULDBLOCK` is another name for `EAGAIN`; they are always the same in the GNU C library.

This error can happen in a few different situations:

- An operation that would block was attempted on an object that has non-blocking mode selected. Trying the same operation again will block until some external condition makes it possible to read, write, or connect (whatever the operation). You can use `select` to find out when the operation will be possible; see section Waiting for Input or Output. **Portability Note:** In many older Unix systems, this condition was indicated by `EWOULDBLOCK`, which was a distinct error code different from `EAGAIN`. To make your program portable, you should check for both codes and treat them the same.
- A temporary resource shortage made an operation impossible. `fork` can return this error. It indicates that the shortage is expected to pass, so your program can try the call again later and it may succeed. It is probably a good idea to delay for a few seconds before trying it again, to allow time for other processes to release scarce resources. Such shortages are usually fairly serious and affect the whole system, so usually an interactive program should report the error to the user and return to its command loop.

Macro: int **EWOULDBLOCK**

In the GNU C library, this is another name for `EAGAIN` (above). The values are always the same, on every operating system.

C libraries in many older Unix systems have `EWOULDBLOCK` as a separate error code.

Macro: int **EINPROGRESS**

An operation that cannot complete immediately was initiated on an object that has non-blocking mode selected. Some functions that must always block (such as`connect`; see section Making a Connection) never return `EAGAIN`. Instead, they return `EINPROGRESS` to indicate that

the operation has begun and will take some time. Attempts to manipulate the object before the call completes return `EALREADY`. You can use the `select` function to find out when the pending operation has completed; see section Waiting for Input or Output.

Macro: int **EALREADY**
An operation is already in progress on an object that has non-blocking mode selected.

Macro: int **ENOTSOCK**
A file that isn't a socket was specified when a socket is required.

Macro: int **EMSGSIZE**
The size of a message sent on a socket was larger than the supported maximum size.

Macro: int **EPROTOTYPE**
The socket type does not support the requested communications protocol.

Macro: int **ENOPROTOOPT**
You specified a socket option that doesn't make sense for the particular protocol being used by the socket. See section Socket Options.

Macro: int **EPROTONOSUPPORT**
The socket domain does not support the requested communications protocol (perhaps because the requested protocol is completely invalid). See sectionCreating a Socket.

Macro: int **ESOCKTNOSUPPORT**
The socket type is not supported.

Macro: int **EOPNOTSUPP**
The operation you requested is not supported. Some socket functions don't make sense for all types of sockets, and others may not be implemented for all communications protocols. In the GNU system, this error can happen for many calls when the object does not support the particular operation; it is a generic indication that the server knows nothing to do for that call.

Macro: int **EPFNOSUPPORT**
The socket communications protocol family you requested is not supported.

Macro: int **EAFNOSUPPORT**
The address family specified for a socket is not supported; it is inconsistent with the protocol being used on the socket. See section Sockets.

Macro: int **EADDRINUSE**
The requested socket address is already in use. See section Socket Addresses.

Macro: int **EADDRNOTAVAIL**

The requested socket address is not available; for example, you tried to give a socket a name that doesn't match the local host name. See section Socket Addresses.

Macro: int **ENETDOWN**

A socket operation failed because the network was down.

Macro: int **ENETUNREACH**

A socket operation failed because the subnet containing the remote host was unreachable.

Macro: int **ENETRESET**

A network connection was reset because the remote host crashed.

Macro: int **ECONNABORTED**

A network connection was aborted locally.

Macro: int **ECONNRESET**

A network connection was closed for reasons outside the control of the local host, such as by the remote machine rebooting or an unrecoverable protocol violation.

Macro: int **ENOBUFS**

The kernel's buffers for I/O operations are all in use. In GNU, this error is always synonymous with `ENOMEM`; you may get one or the other from network operations.

Macro: int **EISCONN**

You tried to connect a socket that is already connected. See section Making a Connection.

Macro: int **ENOTCONN**

The socket is not connected to anything. You get this error when you try to transmit data over a socket, without first specifying a destination for the data. For a connectionless socket (for datagram protocols, such as UDP), you get `EDESTADDRREQ` instead.

Macro: int **EDESTADDRREQ**

No default destination address was set for the socket. You get this error when you try to transmit data over a connectionless socket, without first specifying a destination for the data with `connect`.

Macro: int **ESHUTDOWN**

The socket has already been shut down.

Macro: int **ETOOMANYREFS**

???

Macro: int **ETIMEDOUT**

A socket operation with a specified timeout received no response during the timeout period.

Macro: int **ECONNREFUSED**

A remote host refused to allow the network connection (typically because it is not running the requested service).

Macro: int **ELOOP**

Too many levels of symbolic links were encountered in looking up a file name. This often indicates a cycle of symbolic links.

Macro: int **ENAMETOOLONG**

Filename too long (longer than `PATH_MAX`; see section Limits on File System Capacity) or host name too long (in `gethostname` or `sethostname`; see sectionHost Identification).

Macro: int **EHOSTDOWN**

The remote host for a requested network connection is down.

Macro: int **EHOSTUNREACH**

The remote host for a requested network connection is not reachable.

Macro: int **ENOTEMPTY**

Directory not empty, where an empty directory was expected. Typically, this error occurs when you are trying to delete a directory.

Macro: int **EPROCLIM**

This means that the per-user limit on new process would be exceeded by an attempted `fork`. See section Limiting Resource Usage, for details on the`RLIMIT_NPROC` limit.

Macro: int **EUSERS**

The file quota system is confused because there are too many users.

Macro: int **EDQUOT**

The user's disk quota was exceeded.

Macro: int **ESTALE**

Stale NFS file handle. This indicates an internal confusion in the NFS system which is due to file system rearrangements on the server host. Repairing this condition usually requires unmounting and remounting the NFS file system on the local host.

Macro: int **EREMOTE**

An attempt was made to NFS-mount a remote file system with a file name that already specifies an NFS-mounted file. (This is an error on some operating systems, but we expect it to work properly on the GNU system, making this error code impossible.)

Macro: int **EBADRPC**

???

Macro: int **ERPCMISMATCH**

???

Macro: int **EPROGUNAVAIL**

???

Macro: int **EPROGMISMATCH**

???

Macro: int **EPROCUNAVAIL**

???

Macro: int **ENOLCK**

No locks available. This is used by the file locking facilities; see section File Locks. This error is never generated by the GNU system,

but it can result from an operation to an NFS server running another operating system.

Macro: int **EFTYPE**

Inappropriate file type or format. The file was the wrong type for the operation, or a data file had the wrong format.

On some systems `chmod` returns this error if you try to set the sticky bit on a non-directory file; see section Assigning File Permissions.

Macro: int **EAUTH**

???

Macro: int **ENEEDAUTH**

???

Macro: int **ENOSYS**

Function not implemented. This indicates that the function called is not implemented at all, either in the C library itself or in the operating system. When you get this error, you can be sure that this particular function will always fail with `ENOSYS` unless you install a new version of the C library or the operating system.

Macro: int **ENOTSUP**

Not supported. A function returns this error when certain parameter values are valid, but the functionality they request is not available. This can mean that the function does not implement a particular command or option value or flag bit at all. For functions that operate on some object given in a parameter, such as a file descriptor or a port, it might instead mean that only *that specific object* (file descriptor, port, etc.) is unable to support the other parameters given; different file descriptors might support different ranges of parameter values.

If the entire function is not available at all in the implementation, it returns `ENOSYS` instead.

Macro: int **EILSEQ**

While decoding a multibyte character the function came along an invalid or an incomplete sequence of bytes or the given wide character is invalid.

Macro: int **EBACKGROUND**

In the GNU system, servers supporting the `term` protocol return this error for certain operations when the caller is not in the foreground process group of the terminal. Users do not usually see this error because functions such as `read` and `write` translate it into a `SIGTTIN` or `SIGTTOU` signal. See section Job Control, for information on process groups and these signals.

Macro: int **EDIED**

In the GNU system, opening a file returns this error when the file is translated by a program and the translator program dies while starting up, before it has connected to the file.

Macro: int **ED**

The experienced user will know what is wrong.

Macro: int **EGREGIOUS**

You did **what**?

Macro: int **EIEIO**

Go home and have a glass of warm, dairy-fresh milk.

Macro: int **EGRATUITOUS**

This error code has no purpose.

Macro: int **EBADMSG**
Macro: int **EIDRM**
Macro: int **EMULTIHOP**
Macro: int **ENODATA**
Macro: int **ENOLINK**
Macro: int **ENOMSG**
Macro: int **ENOSR**
Macro: int **ENOSTR**
Macro: int **EOVERFLOW**
Macro: int **EPROTO**
Macro: int **ETIME**

The following error codes are defined by the Linux/i386 kernel. They are not yet documented.

Macro: int **ERESTART**
Macro: int **ECHRNG**
Macro: int **EL2NSYNC**
Macro: int **EL3HLT**
Macro: int **EL3RST**
Macro: int **ELNRNG**
Macro: int **EUNATCH**
Macro: int **ENOCSI**
Macro: int **EL2HLT**
Macro: int **EBADE**
Macro: int **EBADR**
Macro: int **EXFULL**
Macro: int **ENOANO**
Macro: int **EBADRQC**
Macro: int **EBADSLT**
Macro: int **EDEADLOCK**
Macro: int **EBFONT**
Macro: int **ENONET**
Macro: int **ENOPKG**

Macro: int **EADV**
Macro: int **ESRMNT**
Macro: int **ECOMM**
Macro: int **EDOTDOT**
Macro: int **ENOTUNIQ**
Macro: int **EBADFD**
Macro: int **EREMCHG**
Macro: int **ELIBACC**
Macro: int **ELIBBAD**
Macro: int **ELIBSCN**
Macro: int **ELIBMAX**
Macro: int **ELIBEXEC**
Macro: int **ESTRPIPE**
Macro: int **EUCLEAN**
Macro: int **ENOTNAM**
Macro: int **ENAVAIL**
Macro: int **EISNAM**
Macro: int **EREMOTEIO**
Macro: int **ENOMEDIUM**
Macro: int **EMEDIUMTYPE**