# libexplain

## Reference Manual

Peter Miller
*pmiller@opensource.org.au*

.

This document describes libexplain version 1.2
and was prepared 24 July 2013.

## NAME

libexplain − Explain errno values returned by libc functions

## DESCRIPTION

The *libexplain* package provides a library which may be used to explain Unix and Linux system call errors. This will make your application's error messages much more informative to your users.

The library is not quite a drop-in replacement for *strerror*(3), but it comes close. Each system call has a dedicated libexplain function, for example

```
fd = open(path, flags, mode);
if (fd < 0)
{
    fprintf(stderr, "%s\n", explain_open(path, flags, mode));
    exit(EXIT_FAILURE);
}
```

If, for example, you were to try to open `no-such-dir/some-file`, you would see a message like

```
open(pathname = "no-such-dir/some-file", flags = O_RDONLY) failed,
No such file or directory (2, ENOENT) because there is no
"no-such-dir" directory in the current directory
```

The good new is that for each of these functions there is a wrapper function, in this case *explain_open_or_die*(3), that includes the above code fragment. Adding good error reporting is as simple as using a different, but similarly named, function. The library also provides thread safe variants of each explanation function.

Coverage includes 187 system calls and 547 ioctl requests.

### Tutorial Documentation

There is a paper available in PDF format (http://libexplain.sourceforge.net/lca2010/lca2010.pdf) that describes the library and how to use LibExplain. The paper can also be accessed as *explain_lca2010*(1), which also appears in the reference manual (see below).

## HOME PAGE

The latest version of *libexplain* is available on the Web from:

| | | |
|---|---|---|
| URL: | http://libexplain.sourceforge.net/ | |
| File: | index.html | # the libexplain page |
| File: | libexplain.1.2.README | # Description, from the tar file |
| File: | libexplain.1.2.lsm | # Description, LSM format |
| File: | libexplain.1.2.tar.gz | # the complete source |
| File: | libexplain.1.2.pdf | # Reference Manual |

## BUILDING LIBEXPLAIN

Full instructions for building *libexplain* may be found in the *BUILDING* file included in this distribution.

## COPYRIGHT



### Library License

### Non-Library License

*Everything else (all source files that do not constitute the shared library and its include files) are GNU GPL licensed.*



## AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://www.canb.auug.org.au/~millerp/

## RELEASE NOTES

This section details the various features and bug fixes of the various releases. For excruciating and complete detail, and also credits for those of you who have generously sent me suggestions and bug reports, see the *etc/CHANGES.** files.

Coverage includes 187 system calls and 547 ioctl requests.

**Version 1.2 (2013-Mar-15)**

- Explanations are now available for errors reported by the gethostbyname and getrusage system calls.
- Emanuel Haupt <ehaupt@FreeBSD.org> discovered that libexplain coped poorly with different versions of bison emitting code chunks in different orders.

**Version 1.1 (2012-Nov-20)**

- Explanations are now available for errors reported by the *execv*(3), *getresgid*(2), *getresuid*(2), *lchmod*(2), *setgid*(2) *setregid*(2), *setresgid*(2), *setresuid*(2), *setreuid*(2), *setuid*(2) and *utimens*(2) system calls.
- Emanuel Haupt <ehaupt@critical.ch> discovered that the error handling for *shmat*(2) on BSD needed more portability work.
- There are new `explain_filename_from_stream` and `explain_filename_from_fildes` functions to the public API. This gives library clients access to libexplain's idea of the filename.
- Michael Cree <mcree@orcon.net.nz> discovered that there was a problem building libexplain on alpha architecture.
  Debian: Closes: #661440

**Version 1.0 (2012-May-19)**

- Several testing false negative has been fix, concerning EACCES when executed by root.

**Version 0.52 (2012-Mar-04)**

- A false negative in test 76, where Linux security modules change the *rename*(2) semantics.
- A problem on sparc64 has been fixed. Libexplain can now cope with a missing O_LARGEFILE declaration, and yet file flags returned by the kernel have the flag set.
- A build problem on Debian alpha has been fixed, the name of an include file was incorrect.

**Version 0.51 (2012-Jan-26)**

- The *ptrace*(2) support has been improved with more conditionals determined by the ./configure script when building.
  Debian: Closes: #645745

**Version 0.50 (2012-Jan-16)**

• SpepS <spepsforge@users.sf.net> and Eric Smith <eric@brouhaha.com> discovered that _PC_MIN_HOLE_SIZE isn't supported for all Linux. Some more #ifdef was added.

• Several false negatives from tests have been fixed.
Debian: Closes: 654199

• The tarball now includes a libexplain.spec file for building an RPM package using *rpmbuild*(1).

• This change set makes the exe(readlink) string search less particular, so that it works in more cases. In this instance, on Fedora 14.

• Explanations are nowe available for errors reported by the *realpath*(3) system call.

**Version 0.49 (2011-Nov-10)**

- Explanations are now available for errors reported by the *shmctl*(2) system call.
- Some build problems (discovered by the LaunchPad PPA buid farm) have been fixed.

**Version 0.48 (2011-Nov-08)**

- Explanations are now available for errors reported by the *shmat*(2) system call.
- Several build problems on Solaris have been fixed.
- Dagobert Michelsen <dam@opencsw.org> found the test 625 was throwing a false negative in his test environment. It can now cope with stdin being closed.
- Dagobert Michelsen <dam@opencsw.org> discovered that, on Solaris, test false negatives were caused by the need for a space before the width in a "`fmt -w 800`" command.
- Eric Smith <eric@brouhaha.com> discovered that *lsof*(1) could report errors as executable names, when it couldn't read the symlink. These non-results are now filtered out.
- Eric Smith <eric@brouhaha.com> discovered three false negatives from tests of the *kill*(2) system call.
- Better explanations are now available when a user attempts to execute a directory.

**Version 0.47 (2011-Sep-27)**

- Explanations are now available for errors reported by the *setsid*(2) system call.
- The Ubuntu PPA build farm found several Hardy build problems. These have been fixed.
- Code has been added to detect those cases where a file descriptor may be open for reading and writing, but the I/O stream it is accessed by is only open for one of them.
- Code has been added to cope with false negatives when *lsof*(1) is not as helpful as could be desired.
- Michael Bienia <geser@ubuntu.com> discovered a build problem with the SIOCSHWTSTAMP ioctl request, and sent a patch.

**Version 0.46 (2011-Aug-24)**

- LibExplain has been ported to Solaris 8, 9 and 10. My thanks to Dagobert Michelsen and `http://opencsw.org/` for assistance with this port.
- Several more Linux *ioctl*(2) requests are supported.
- A segfault has been fixed in the output tee filter when handling exit.

**Version 0.45 (2011-Jul-17)**

- Dagobert Michelsen <dam@opencsw.org> discoversed several build problems on OpenSolaris; these have been fixed.
- Explanations are now available for errors reported by the Linux *ioctl*(2) V4L1 system calls.

**Version 0.44 (2011-Jul-03)**

- Several build problem to do with older Linux kernels have been fixed.

**Version 0.42 (2011-Jul-02)**

- Explanations are now available for errors reported by the V4L2 ioctl requests.
- The Debian package no longer installs the libtool *.la file.
  Debian: Closes: 621621
- The call arguments printed for ioctl(2) now include the type of the third argument.
- The error messages now include more information about block and character special devices, when printing file types.

**Version 0.42 (2011-May-26)**

- This change set adds an “ldconfig” hint to the BUILDING instructions. My thanks to Blake McBride <blake@arahant.com> for this suggestion.
- Emanuel Haupt <ehaupt@critical.ch> reported several problems building libexplain on FreeBSD. These have been fixed.

**Version 0.41 (2011-Mar-15)**

- There were some C++ keywords in the unclude files, which caused problems for C++ users. They have been replaced.
- Explanations are now availaible for errors reported by the *getpgid*(2), *getpgrp*(2), *ptrace*(2), *setgpid*(2) and *setpgrp*(2) system calls.

**Version 0.40 (2010-Oct-05)**

- The code now builds and tests successfully on FreeBSD.
- Explanations are now available for errors reported by the *calloc*(3) and *poll*(2) system calls.

**Version 0.39 (2010-Sep-12)**

- A build problem has been fixed on Ubuntu Hardy, a number of symbols are absent from older versions of <linux/cdrom.h>, conditional code has been added for them.
- A bug has been fixed in one of the documentation files, it was missing the conditional around the `.XX` macro, causing *rpmlint*(1) and *lintian*(1) to complain.

**Version 0.38 (2010-Sep-08)**

- Some build problems on Fedora 13 have been fixed.

**Version 0.37 (2010-Aug-27)**

- The library source files are supposed to be LGPL, however over 1000 of them were GPL (about 20%). This has been fixed.
- A couple of problems building on Fedora 13 have been fixed.

**Version 0.36 (2010-Aug-25)**

- Several false negative reported by tests on the Linux “alpha” and “ia64” architectures have been fixed.

**Version 0.35 (2010-Aug-15)**

- A number of falve negatives from tests have been fixed, primarily due to random differences between Linux architectures.
- The BUILDING document goes into more detail about things that can cause false negatives when testing.
- The man pages have been fixed so that they no longer contain unescaped hyphen characters, as warned about by the *lintian*(1) program.

**Version 0.34 (2010-Aug-07)**

- Another test 33 false negative has been fixed.
- There is a new “hanging-indent” option, that can be set from the `EXPLAIN_OPTION` environment variable. It defaults to zero for backwards compatibility. Applications may set it using the *explain_option_hanging_indent_set*(3) function.

**Version 0.33 (2010-Jul-04)**

- A number of testing false negatives (found by the Debian build farm) have been fixed.
- There are new *explain_output_error*(3) and *explain_output_error_and_die*(3) functions for printing formatted error messages.
- Some systems have *mmap*(2) report `(void*)(-1)` instead of NULL for errors. This is now understood.

**Version 0.32 (2010-Jun-22)**

- Explanations are now available for errors reported by the *mmap*(2), *munmap*(2) and *utimes*(2) system calls.
- A number of false negatives for tests on some less common architectures have been fixed.
- Some build problems relating to *ioctl*(2) support have been fixed.
- A bug has been fixed in the `libexplain/output.h` file, it was missing the C++ insulation.

**Version 0.31 (2010-May-01)**

- A number of build problems have been fixed.

**Version 0.30 (2010-Apr-28)**
- Several test false negatives have been fixed, on various Debian architectures.

**Version 0.29 (2010-Apr-25)**
- A number of build problems, discovered by the Debian build farm, have been fixed. Who would of thought that there could be some much inconsistency between Linux architectures?

**Version 0.28 (2010-Apr-19)**
- Several architecture-specific build problems, found by the Debian build farm, have been fixed.

**Version 0.27 (2010-Apr-17)**
- Several architecture-specific build problems, found by the Debian build farm, have been fixed.

**Version 0.26 (2010-Apr-06)**
- A build problem has been fixed on systems where `va_list` is not compatible with `const void *`
- This change set removes the unused-result warning form *explain_lseek_or_die*(3), because it is very common to ignore the result.
- Explanations are now available for errors reported by the *socketpair*(2) system call.

**Version 0.25 (2010-Mar-22)**
- Portability of the code has been improved.
- The *explain*(3) man page now mentions AC_SYS_LARGEFILE in the building requirements.
- Coverage now includes the *fprintf*(3), *printf*(3), *snprintf*(3), *sprintf*(3), *vfprintf*(3), *vprintf*(3), *vsnprintf*(3) and *vsprintf*(3) system calls.

**Version 0.24 (2010-Mar-03)**
- It is now possible to redirected libexplain output. For example, it is now possible to redirect all output to *syslog*(3).
- Coverage now includes the *fstatvfs*(2) and *statvfs*(2) system call.
- A number of problems found while building and testing on Solaris have been fixed.

**Version 0.23 (2010-Feb-21)**
- It turns out that on alpha architecture, you can't disambiguate the FIBMAP vs BMP_IOCTL case in the pre-processor. The code now uses a disambiguate function. This problem was discovered by the Debian build farm.

**Version 0.22 (2010-Feb-12)**
- This change set fixes a false negative found by the Debian automated build system.

**Version 0.21 (2010-Feb-09)**

- Explanations are now available for errors reported by the *fpurge*(3), *getw*(3) and *putw*(3) system calls.
- Some build problems have been fixed.

**Version 0.20 (2010-Jan-20)**

- Several lintian warnings relating to the man pages have been fixed.
- The LIBEXPLAIN_OPTIONS environment variable now understands a new symbolic-mode-bits=true option. It defaults to false, for shorter error explanations.
- There is a new *explain_lca2010*(1) man page. This is a gentle introduction to libexplain, and the paper accompanying my LCA 2010 talk.
- When process ID (pid) values are printed, they are now accompanied by the name of the process executable, when available.
- Numerous build bugs and niggles have been fixed.
- Explanations are now available for errors reported by the *execlp*(3), *fdopendir*(3), *feof*(3), *fgetpos*(3), *fputs*(3), *fseek*(3), *fsetpos*(3), *fsync*(2), *ftell*(3), *mkdtemp*(3), *mknod*(2), *mkostemp*(3), *mkstemp*(3), *mktemp*(3), *putenv*(3), *puts*(3), *raise*(3), *setbuf*(3), *setbuffer*(3), *setenv*(3), *setlinebuf*(3), *setvbuf*(3), *stime*(2), *tempnam*(3), *tmpfile*(3), *tmpnam*(3), *ungetc*(3), *unsetenv*(3) and *vfork*(2) system calls.
- The ioctl requests from linux/sockios.h, linux/ext2_fs.h, linux/if_eql.h, PPP, linux/lp.h, and linux/vt.h are now understood. Several of the ioctl explanations have been improved.

**Version 0.19 (2009-Sep-07)**

- The ioctl requests from linux/hdreg.h are now understood.
- Some build problems on Debian Lenny have been fixed.

**Version 0.18 (2009-Sep-05)**

- More ioctl requests are understood.
- Explanations are now available for errors reported by the *tcsendbreak*(3), *tcsetattr*(3), *tcgetattr*(3), *tcflush*(3), *tcdrain*(3), system calls.

**Version 0.17 (2009-Sep-03)**

- Explanations are now available for errors reported by the *telldir*(3) system call.
- A number of Linux build problems have been fixed.
- Explanations for a number of corner-cases of the *open*(2) system call have been improved, where flags values interact with file types and mount options.
- A number of BSD build problems have been fixed.
- More *ioctl*(2) commands are understood.
- A bug has been fixed in the way absolute symbolic links are processed by the path_resolution code.

**Version 0.16 (2009-Aug-03)**

- The EROFS and ENOMEDIUM explanations now greatly improved.
- A number of build problems and false negatives have been fixed on x86_64 architecture.
- The Linux floppy disk and CD-ROM ioctl requests are now supported.
- Explanations are now available for the errors reported by the *getdomainname*(2), *readv*(2), *setdomainname*(2), *ustat*(2) and *writev*(2) system calls.

**Version 0.15 (2009-Jul-26)**

- A number of build errors and warnings on amd64 have been fixed. The problems were only detectable on 64-bit systems.

**Version 0.14 (2009-Jul-19)**

- Coverage now includes another 29 system calls: *accept4*(2), *acct*(2), *adjtime*(3), *adjtimex*(2), *chroot*(2), *dirfd*(3), *eventfd*(2), *fflush*(3), *fileno*(3), *flock*(2), *fstatfs*(2), *ftime*(3), *getgroups*(2), *gethostname*(2), *kill*(2), *nice*(2), *pread*(2), *pwrite*(2), *sethostname*(2), *signalfd*(2), *strdup*(3), *strtod*(3), *strtof*(3), *strtol*(3), *strtold*(3), *strtoll*(3), *strtoul*(3), *strtoull*(3), and *timerfd_create*(2). A total of 110 system calls are now supported
- The ./configure script no longer demands *lsof*(1). The Linux libexplain code doesn't need *lsof*(1). On systems not supported by *lsof*(1), the error messages aren't quite as useful, but libexplain still works.
- There is now an `explain_*_on_error` function for each system call, each reports errors but still returns the original return value to the caller.

**Version 0.13 (2009-May-17)**

- The web site now links to a number of services provided by SourceForge.
- Several problems have been fixed with compiling libexplain on 64-bit systems.

**Version 0.12 (2009-May-04)**

- A build problem has been fixed on hosts that didn't need to do anything special for large file support.

**Version 0.11 (2009-Mar-29)**

- The current directory is replaced in messages with an absolute path in cases where the user's idea of the current directory may differ from that of the current process.

**Version 0.10 (2009-Mar-24)**

- The name prefix on all of the library functions has been changed from "libexplain_" to just "explain_". This was *the* most requested change. You will need to change your code and recompile. Apologies for the inconvenience.

**Version 0.9 (2009-Feb-27)**

- Two false negatives in the tests have been fixed.
- The ./configure script now explicitly looks for *bison*(1), and complains if it cannot be found.
- The *socket*(7) address family is now decoded.

**Version 0.8 (2009-Feb-14)**

- A problem with the Debian packaging has been fixed.
- The decoding of IPv4 sockaddr structs has been improved.

**Version 0.7 (2009-Feb-10)**

- Coverage has been extended to include *getsockopt*(2), *getpeername*(2), *getsockname*(2) and *setsockopt*(2).
- Build problems on Debian Sid have been fixed.
- More magnetic tape ioctl controls, from operating systems other than Linux, have been added.

**Version 0.6 (2009-Jan-16)**

- Coverage has been extended to include *execvp*(3), *ioctl*(2), *malloc*(3), *pclose*(3), *pipe*(2), *popen*(3) and *realloc*(3) system calls.
- The coverage for *ioctl*(2) includes linux console controls, magnetic tape controls, socket controls, and terminal controls.
- A false negative from test 31 has been fixed.

**Version 0.5 (2009-Jan-03)**

- A build problem on Debian sid has been fixed.
- There is a new *explain_system_success*(3) function, that performs all that *explain_system_success_or_die*(3) performs, except that it does not call *exit*(2).
- There is more i18n support.
- A bug with the *pkg-config*(1) support has been fixed.

**Version 0.4 (2008-Dec-24)**

- Coverage now includes *accept*(2), *bind*(2), *connect*(2), *dup2*(2), *fchown*(2), *fdopen*(3), *fpathconf*(2), *fputc*(2), *futimes*(2), *getaddrinfo*(2), *getcwd*(2), *getrlimit*(2), *listen*(2), *pathconf*(2), *putc*(2), *putchar*(2), *select*(2).
- Internationalization has been improved.
- The thread safety of the code has been improved.
- The code is now able to be compiled on OpenBSD. The test suite still gives many false negatives, due to differences in *strerror*(3) results.

**Version 0.3 (2008-Nov-23)**

- Cover has been extended to include *closedir*(3), *execve*(2), *ferror*(3), *fgetc*(3), *fgets*(3), *fork*(2), *fread*(3), *getc*(3), *gettimeofday*(2), *lchown*(2), *socket*(2), *system*(3), *utime*(2), *wait3*(2), *wait4*(2), *wait*(2), *waitpid*(2),
- More internationalization support has been added.
- A bug has been fixed in the C++ insulation.

**Version 0.2 (2008-Nov-11)**

- Coverage now includes *chmod*(2), *chown*(2), *dup*(2), *fchdir*(2), *fchmod*(2), *fstat*(2), *ftruncate*(2), *fwrite*(3), *mkdir*(2), *readdir*(3), *readlink*(2), *remove*(3), *rmdir*(2) and *truncate*(2).
- The *lsof*(1) command is used to obtain supplementary file information on those systems with limited `/proc` implementations.
- The explanations now understand Linux capabilities.

**Version 0.1 (2008-Oct-26)**

First public release.

## NAME

How to build libexplain

## SPACE REQUIREMENTS

You will need about 6MB to unpack and build the *libexplain* package. Your milage may vary.

## BEFORE YOU START

There are a few pieces of software you may want to fetch and install before you proceed with your installation of libexplain

libcap
: Linux needs libcap, for access to capabilities.
ftp://ftp.kernel.org/pub/linux/libs/security/linux−privs/kernel−2.2/

lsof
: For systems with inadequate or non-existent /proc facilities, and that includes *BSD and MacOS X, the *lsof*(1) program is needed to obtain supplementary information about open file descriptors. However, if *lsof*(1) is not supported on your operating system, libexplain will still work, but some useful information (such as translating file descriptors into the name of the open file) will be absent from error explanations.

  ftp://lsof.itap.purdue.edu/pub/tools/unix/lsof/
  http://people.freebsd.org/~abe/

  You **must** have *lsof*(1) installed on *BSD and Solaris, otherwise the test suite will generate staggering numbers of false negatives. It will produce less informative error messages, too.

  Supported systems include: Free BSD, HP/UX, Linux, Mac OS X, NetBSD, Open BSD, Solaris, and several others.

GNU libtool
: The libtool program is used to build shared libraries. It understands the neccesary, weird and wonderful compiler and linker tricks on many weird and wonderful systems.
http://www.gnu.org/software/libtool/

bison
: The bison program is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar.
http://www.gnu.org/software/bison/

GNU Groff
: The documentation for the *libexplain* package was prepared using the GNU Groff package (version 1.14 or later). This distribution includes full documentation, which may be processed into PostScript or DVI files at install time − if GNU Groff has been installed.

GCC
: You may also want to consider fetching and installing the GNU C Compiler if you have not done so already. This is not essential. libexplain was developed using the GNU C compiler, and the GNU C libraries.

The GNU FTP archives may be found at `ftp.gnu.org`, and are mirrored around the world.

## SITE CONFIGURATION

The **libexplain** package is configured using the *configure* program included in this distribution.

The *configure* shell script attempts to guess correct values for various system-dependent variables used during compilation, and creates the *Makefile* and *libexplain/config.h* files. It also creates a shell script *config.status* that you can run in the future to recreate the current configuration.

Normally, you just *cd* to the directory containing *libexplain*'s source code and then type

```
$ ./configure --prefix=/usr
...lots of output...
$
```

If you're using *csh* on an old version of System V, you might need to type

```
% sh configure --prefix=/usr
...lots of output...
```

```
%
```

instead, to prevent *csh* from trying to execute *configure* itself.

Running *configure* takes a minute or two. While it is running, it prints some messages that tell what it is doing. If you don't want to see the messages, run *configure* using the quiet option; for example,

```
$ ./configure --prefix=/usr --quiet
$
```

To compile the **libexplain** package in a different directory from the one containing the source code, you must use a version of *make* that supports the VPATH variable,such as *GNU make*, *cd* to the directory where you want the object files and executables to go and run the *configure* script. The *configure* script automatically checks for the source code in the directory that *configure* is in and in .IR .. (the parent directory). If for some reason *configure* is not in the source code directory that you are configuring, then it will report that it can't find the source code. In that case, run *configure* with the option `--srcdir=`*DIR*, where *DIR* is the directory that contains the source code.

By default, *configure* will arrange for the *make install* command to install the **libexplain** package's files in */usr/local/bin*, */usr/local/lib*, */usr/local/include*, and */usr/local/man*. There are options which allow you to control the placement of these files.

`--prefix=`*PATH*
: This specifies the path prefix to be used in the installation. Defaults to */usr/local* unless otherwise specified.

`--exec-prefix=`*PATH*
: You can specify separate installation prefixes for architecture-specific files files. Defaults to *${prefix}* unless otherwise specified.

`--bindir=`*PATH*
: This directory contains executable programs. On a network, this directory may be shared between machines with identical hardware and operating systems; it may be mounted read-only. Defaults to *${exec_prefix}/bin* unless otherwise specified.

`--mandir=`*PATH*
: This directory contains the on-line manual entries. On a network, this directory may be shared between all machines; it may be mounted read-only. Defaults to *${prefix}/man* unless otherwise specified.

*configure* ignores most other arguments that you give it; use the `--help` option for a complete list.

On systems that require unusual options for compilation or linking that the *libexplain* package's *configure* script does not know about, you can give *configure* initial values for variables by setting them in the environment. In Bourne-compatible shells, you can do that on the command line like this:

```
$ CC='gcc -ansi' LIBS=-lposix ./configure
...lots of output...
$
```

Here are the *make* variables that you might want to override with environment variables when running *configure*.

Variable: CC
: C compiler program. The default is *gcc*.

Variable: CPPFLAGS
: Preprocessor flags, commonly defines and include search paths. Defaults to empty. It is common to use `CPPFLAGS=-I/usr/local/include` to access other installed packages.

Variable: INSTALL
: Program to use to install files. The default is *install*(1) if you have it, *cp*(1) otherwise.

Variable: LIBS
: Libraries to link with, in the form `-l`*foo* `-l`*bar*. The *configure* script will append to this, rather than replace it. It is common to use `LIBS=-L/usr/local/lib` to access other installed

packages.

If you need to do unusual things to compile the package, the author encourages you to figure out how *configure* could check whether to do them, and mail diffs or instructions to the author so that they can be included in the next release.

## BUILDING LIBEXPLAIN

All you should need to do is use the

```
$ make
...lots of output...
$
```

command and wait. This can take a long time, as there are a few thousand files to be compiled.

You can remove the program binaries and object files from the source directory by using the

```
$ make clean
...lots of output...
$
```

command. To remove all of the above files, and also remove the *Makefile* and *libexplain/config.h* and *config.status* files, use the

```
$ make distclean
...lots of output...
$
```

command.

The file *etc/configure.ac* is used to create *configure* by a GNU program called *autoconf*. You only need to know this if you want to regenerate *configure* using a newer version of *autoconf*.

## TESTING LIBEXPLAIN

The *libexplain* package comes with a test suite. To run this test suite, use the command

```
$ make sure
...lots of output...
Passed All Tests
$
```

The tests take a fraction of a second each, with most very fast, and a couple very slow, but it varies greatly depending on your CPU.

If all went well, the message

```
Passed All Tests
```

should appear at the end of the make.

### Sources of False Negatives

There are a number of factors that can cause tests to fail unnecessarily.

Root
: You will get false negatives if you run the tests as root.

Architecture
: Some errors move around depending on architecture (sparc *vs* x86 *vs* s390, *etc*). Some even move around due to different memory layout for 32-bit *vs* 64-bit, for the same processor family. For example, when testing EFAULT explanations.

strerror
: Different systems have different *strerror*(3) implementations (the numbers vary, the texts vary, the existence varies, *etc*). This can even be incompatible across Linux architectures when ABI compatibility was the goal, *e.g.* sparc *vs* i386.

ioctl
: There are (at least) three inconsistent implementations of ioctl request macros, all incompatible, depending on Unix vendor. They also vary on Linux, depending on architecture, for ABI compatibility reasons.

Environment
: Some tests are difficult because the build-and-test environment can vary widely. Sometimes it's a chroot, sometimes it's a VM, sometimes it's fakeroot, sometimes it really is running as root. All

these affect the ability of the library to probe the system looking for the proximal cause of the error, *e.g.* ENOSPC or EROFS. This often results in 2 or 4 or 8 explanations of an error, depending on what the library finds, *e.g.* existence of useful information in the mount table, or not.

Mount Table
: If you run the tests in a chroot jail build environment, maybe with bind mounts for the file systems, it is necessary to make sure */etc/mtab* (or equivalent) has sensable contents, otherwise some of the path resolution tests will return false negatives.

/proc
: If your system has a completely inadequate `/proc` implementation (including, but not limited to: *BSD, Mac OS X, and Solaris) or no `/proc` at all, **and** you have not installed the *lsof*(1) tool, then large numbers of tests will return false negatives.

As these problem have occured, many of the tests have been enhanced to cope, but not all false negative situations have yet been discovered.

## INSTALLING LIBEXPLAIN

As explained in the *SITE CONFIGURATION* section, above, the *libexplain* package is installed under the */usr/local* tree by default. Use the `--prefix=`*PATH* option to *configure* if you want some other path. More specific installation locations are assignable, use the `--help` option to *configure* for details.

All that is required to install the *libexplain* package is to use the

```
# make install
...lots of output...
#
```

command. Control of the directories used may be found in the first few lines of the *Makefile* file and the other files written by the *configure* script; it is best to reconfigure using the *configure* script, rather than attempting to do this by hand.

**Note:** if you are doing a manual install (as opposed to a package build) you will also need to run the

```
# ldconfig
#
```

command. This updates where the system thinks all the shared libraries are. And since we just installed one, this is a good idea.

## GETTING HELP

If you need assistance with the *libexplain* package, please do not hesitate to contact the author at

```
Peter Miller <pmiller@opensource.org.au>
```

Any and all feedback is welcome.

When reporting problems, please include the version number given by the

```
$ explain -version
explain version 1.2.D001
...warranty disclaimer...
$
```

command. Please do not send this example; run the program for the exact version number.

## COPYRIGHT



## AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://www.canb.auug.org.au/~millerp/

## NAME

new system call − How to add a new system call to libexplain

## DESCRIPTION

Adding a new system call to libexplain is both simple and tedious.

In this example, the system call is called *example*, and takes two arguments, *pathname* and *flags*.

```
example(const char *pathname, int flags);
```

The libexplain library presents a C interface to the user, and explains the C system calls. It tries to avoid dynamic memory, and has several helper functions and structures to make this simpler.

### Naming Conventions

In general, one function per file. This gives the static linker more opportunity to leave things out, thus producing smaller executables. Exceptions to make use of `static` common functions are acceptable. No savings for shared libraries, of course.

Functions that write their output into a *explain_string_buffer_t* via the `explain_string_buffer_*` functions, all have a filename of `libexplain/buffer/`*something*.

Functions that write their output to a *message*, *message_size* pair have a `message` path component in their file name.

Functions that accept an *errno* value as an argument have an `errno` path component in their file name, called `errnum`. If a function has both a buffer and an errno, the buffer comes first, both in the argument list, and the file's name. If a function has both a message+size and an errno, the message comes first, both in the argument list, and the file's name.

## MODIFIED FILES

Note that the *codegen* command does most of the work for you. Pass it the function prototype (in single quotes) and it will do most of the work.

```
$ bin/codegen 'example(const char *pathname, int flags);'
creating catalogue/example
$
```

then you mast edit the `catalogue/example` file to make any adjustment necessary. This file is then used to do the boring stuff:

```
$ bin/codegen example
creating explain/syscall/example.c
creating explain/syscall/example.h
creating libexplain/buffer/errno/example.c
creating libexplain/buffer/errno/example.h
creating libexplain/example.c
creating libexplain/example.h
creating libexplain/example_or_die.c
creating man/man3/explain_example.3
creating man/man3/explain_example_or_die.3
creating test_example/main.c
modify explain/syscall.c
modify libexplain/libexplain.h
modify man/man1/explain.1
modify man/man3/explain.3
$
```

All of these files have been added to the Aegis change set. Edit the last 4 to place the appended line in their correct positions within the files, respecting the symbol sort ordering of each file.

### libexplain/libexplain.h

The `libexplain/libexplain.h` include file defines the user API. It, and any files it includes, are installed into `$(prefix)/include` by *make install*.

This file needs another include line. This means that the entire API is available to the user as a single

include directive.

```
#include <libexplain/example.h>
```

This file is also used to decide which files are installed by the *make install* command.

Take care that none of those files, directly or indirectly, wind up including `libexplain/config.h` which is generated by the *configure* script, and has **no** namespace protection.

This means you can't `#include <stddef.h>`, or use any of the types it defines, because on older systems *configure* works quite hard to cope with its absence. Ditto `<unistd.h>` and `<sys/types.h>`.

**explain/main.c**
Include the include file for the new function, and add the function to the table.

**man/man1/explain.1**
Add a description of the new system call.

**man/man3/libexplain.3**
Add your new man pages, man/man3/explain_*example*.3 and man/man3/explain_*example*_or_die.3, to the list. Keep the list sorted.

## NEW FILES

Note that the *codegen* command does most of the work for you. Pass it the function prototype (in single quotes) and it will do most of the work.

**libexplain/buffer/errno/*example*.c**
The central file for adding a new example is `libexplain/buffer/errno/`*example*`.c` Which defines a function

```
void explain_buffer_errno_example(explain_string_buffer_t *buffer,
int errnum, const char *pathname, int flags);
```

The `errnum` argument holds the *errno* value. Note that calling *errno* usually has problems because many systems have *errno* as a macro, which makes the compiler barf, and because there are times you want access to the global *errno*, and having it shadowed by the argument is a nuisance.

This function writes its output into the buffer via the `explain_string_buffer_printf`, *etc*, functions. First the argument list is reprinted.

The `explain_string_buffer_puts_quoted` function should be used to print pathnames, because it uses full C quoting and escape sequences.

If an argument is a file descriptor, it should be called *fildes*, short for "file descriptor". On systems capable of it, the file descriptor can be mapped to a pathname using the `explain_buffer_fildes_to_pathname` function. This makes explanations for system calls like *read* and *write* much more informative.

Next comes a switch on the errnum value, and additional explanation is given for each errno value documented (or sometimes undocumented) for that system call. Copy-and-paste of the man page is often useful as a basis for the text of the explanation, but be sure it is open source documentation, and not Copyright proprietary text.

Don't forget to check the existing `libexplain/buffer/e*.h` files for pre-canned explanations for common errors. Some pre-canned explanations include

| | |
|---|---|
| EACCES | explain_buffer_eacces |
| EADDRINUSE | explain_buffer_eaddrinuse |
| EAFNOSUPPORT | explain_buffer_eafnosupport |
| EBADF | explain_buffer_ebadf |
| EFAULT | explain_buffer_efault |
| EFBIG | explain_buffer_efbig |
| EINTR | explain_buffer_eintr |
| EINVAL | explain_buffer_einval_vague, *etc* |

| | |
|---|---|
| EIO | explain_buffer_eio |
| ELOOP | explain_buffer_eloop |
| EMFILE | explain_buffer_emfile |
| EMLINK | explain_buffer_emlink |
| ENAMETOOLONG | explain_buffer_enametoolong |
| ENFILE | explain_buffer_enfile |
| ENOBUFS | explain_buffer_enobufs |
| ENOENT | explain_buffer_enoent |
| ENOMEM | explain_buffer_enomem |
| ENOTCONN | explain_buffer_enotconn |
| ENOTDIR | explain_buffer_enotdir |
| ENOTSOCK | explain_buffer_enotsock |
| EROFS | explain_buffer_erofs |
| ETXTBSY | explain_buffer_etxtbsy |
| EXDEV | explain_buffer_exdev |

**libexplain/buffer/errno/example.h**
This file holds the function prototype for the above function definition.

**libexplain/example.h**
The file contains the user visible API for the *example* system call. There are five function prototypes declared in this file:

`void explain_`*example*`_or_die(`*const char *pathname, int flags*`);`
`void explain_`*example*`(` *const char *pathname, int flags*`);`
`void explain_errno_`*example*`(int errnum,` *const char *pathname, int flags*`);`
`void explain_message_`*example*`(const char *message, int message_size,` *const char *pathname, int flags*`);`
`void explain_message_errno_`*example*`(const char *message, int message_size, int errnum,` *const char *pathname, int flags*`);`

The function prototypes for these appear in the `libexplain/`*example*`.h include file.`

Each function prototype shall be accompanied by thorough Doxygen style comments. These are extracted and placed on the web site.

The buffer functions are **never** part of the user visible API.

**libexplain/***example***_or_die.c**
One function per file, `explain_`*example*`_or_die` in this case. It simply calls *example* and then, if fails, `explain_`*example* to print why, and then exit(EXIT_FAILURE).

**libexplain/example.c**
One function per file, `explain_`*example* in this case. It simply calls `explain_errno_`*example* to pass in the global *errno* value.

**libexplain/errno/example.c**
One function per file, `explain_errno_`*example* in this case. It calls `explain_message_errno_`*example*, using the `<libexplain/global_message_buffer.h>` to hold the string.

**libexplain/message/example.c**
One function per file, `explain_message_`*example* in this case. It simply calls `explain_message_errno_`*example* to pass in the global *errno* value.

**libexplain/message/errno/example.c**
One function per file, `explain_message_errno_`*example* in this case. It declares and initializes a `explain_string_buffer_t` instance, which ensures that the message buffer will not be exceeded, and passes that buffer to the `explain_buffer_errno_`*example* function.

**man/man3/explain_example.3**
This file also documents the error explanations functions, except `explain_`*example*`_or_dir`. Use the same text as you did in `libexplain/`*example*`.h`

**man/man3/explain_example_or_die.3**
This file also documents the helper function. Use the same text as you did in `libexplain/`*example*`.h`

**explain/example.c**
Glue to turn the command line into arguments to a call to `explain_`*example*

**explain/example.h**
Function prototype for the above.

**test_example/main.c**
This program should call `explain_`*explain*`_or_die`.

## NEW IOCTL REQUESTS

Each different *ioctl*(2) request is, in effect, yet another system call. Except that they all have appallingly bad type safety. I have seen fugly C++ classes with less overloading than *ioctl*(2).

libexplain/iocontrol/request_by_number.c
This file has one include line for each *ioctl*(2) request. There is a `table` array that contains a pointer to the explain_iocontrol_t variable declared in the include file (see next). Keep both sets of lines sorted alphabetically, it makes it easier to detect duplicates.

libexplain/iocontrol/*name*.h
Where *name* is the name of the *ioctl*(2) request in lower case. This declares an global const variable describing how to handle it.

libexplain/iocontrol/*name*.c
This defines the above global variable, and defines any static glue functions necessary to print a representation of it. You will probably have to read the kernel source to discover the errors the ioctl can return, and what causes them, in order to write the explanation function; they are almost never described in the man pages.

## TESTS

Write at least one separate test for each case in the errnum switch.

## Debian Notes

You can check that the Debian stuff builds by using

```
apt-get install pbuilder
pbuiler create
pbuilder login
```

now copy the files from *web-site/debian/* into the chroot

```
cd libexplain-*
dpkg-checkbuilddeps
apt-get install what dpkg-checkbuilddeps said
apt-get install devscripts
debuild
```

This should report success.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain − explain system call error messages

## SYNOPSIS

**explain** [ *option* ... ] *function* [ *argument* ... ]

**explain −−version**

## DESCRIPTION

The explain command is used to decode an error return read from an *strace*(1) listing, or silimar. Because thisd is being deciphered in a different process than the orginal, the results will be less accurate than if the program itself were to use *libexplain*(3).

### Functions

The functions understood include:

accept *fildes addr addrlen*
: The *accept*(2) system call.

accept4 *fildes* [[ *sock_addr sock_addr_size* ] *flags* ]
: The *accept4*(2) system call.

access *pathname*
: The *access*(2) system call.

acct *pathname*
: The *acct*(2) system call.

adjtime *delta olddelta*
: The *adjtime*(2) system call.

adjtimex *data*
: The *adjtimex*(2) system call.

bind *fildes addr sockaddr_size*
: The *bind*(2) system call.

calloc *nmemb size*
: The *calloc*(3) system call.

chdir *pathname*
: The *chdir*(2) system call.

chmod *pathname permission-mode*
: The *chmod*(2) system call.

chown *pathname owner group*
: The *chown*(2) system call.

chroot *pathname*
: The *chroot*(2) system call.

close *fildes*
: The *close*(2) system call.

closedir *dir*
: The *closedir*(3) system call.

connect *fildes serv_addr serv_addr_size*
: The *connect*(2) system call.

creat *pathname* [ *permission-mode*
: The *creat*(2) system call.

dirfd *dir* The *dirfd*(3) system call.

dup *fildes*
The *dup*(2) system call.

dup2 *oldfd newfd*
The *dup2*(2) system call.

eventfd *initval flags*
The *eventfd*(2) system call.

execlp *pathname arg*...
The *execlp*(3) system call.

execv *pathname argv*
The *execv*(3) system call.

execve *pathname arg*...
The *execve*(2) system call.

execvp *pathname arg*...
The *execvp*(3) system call.

fchdir *pathname*
The *fchdir*(2) system call.

fchown *fildes owner group*
The *fchown*(2) system call.

fclose *fp* The *fclose*(3) system call.

fcntl *fildes command* [ *arg* ]
The *fcntl*(2) system call.

fdopen *fd mode*
The *fdopen*(3) system call.

fdopendir *fildes*
The *fdopendir*(3) system call.

feof *fp* The *feof*(3) system call.

ferror *fp* The *ferror*(3) system call.

fflush *fp* The *fflush*(3) system call.

fgetc *fp* The *fgetc*(3) system call.

fgetpos *fp pos*
The *fgetpos*(3) system call.

fgets *data data_size fp*
The *fgets*(3) system call.

fileno *fp* The *fileno*(3) system call.

flock *fildes command*
The *flock*(2) system call.

fork The *fork*(2) system call.

fpathconf *fildes name*
The *fpathconf*(3) system call.

fpurge *fp*
The *fpurge*(3) system call.

fread *ptr size nmemb fp*
The *fread*(3) system call.

fopen *pathname mode*
The *fopen*(2) system call. The *pathname* argument may need to be quoted to insulate white space and punctuation from the shell. The *mode* argument (a textual equivalent of the *open* system call's *flags* argument). See *fopen*(3) for more information.

fputc *c* [ *fp* ]
The *fputc*(3) system call.

fputs *s fp*
The *fputs*(3) system call.

freopen *pathname flags fp*
The *freopen*(3) system call.

fseek *fp offset whence*
The *fseek*(3) system call.

fsetpos *fp pos*
The *fsetpos*(3) system call.

fstat *pathname*
The *fstat*(2) system call.

fstatfs *fildes data*
The *fstatfs*(2) system call.

fstatvfs *fildes data*
The *fstatvfs*(2) system call.

fsync *fildes*
The *fsync*(2) system call.

ftell *fp* The *ftell*(3) system call.

ftime *tp* The *ftime*(3) system call.

ftruncate *fildes length*
The *ftruncate*(2) system call.

futimes *fildes tv[0] tv[1]*
The *futimes*(3) system call.

getc *fp* The *getc*(3) system call.

getchar The *getchar*(3) system call.

getcwd *buf size*
The *getcwd*(2) system call.

getdomainname *data data_size*
The *getdomainname*(2) system call.

getgroups *data_size data*
The *getgroups*(2) system call.

gethostbyname *name*
The *gethostbyname*(3) system call.

gethostname [ *data data_size* ]
The *gethostname*(2) system call.

getpeername *fildes sock_addr sock_addr_size*
The *getpeername*(2) system call.

getpgid *pid*
The *getpgid*(2) system call.

getpgrp *pid*
The *getpgrp*(2) system call.

getresgid *rgid egid sgid*
The *getresgid*(2) system call.

getresuid *ruid euid suid*
The *getresuid*(2) system call.

getrlimit *resource rlim*
The *getrlimit*(2) system call.

getrusage *who usage*
The *getrusage*(2) system call.

getsockname *fildes* [ *sock_addr* [ *sock_addr_size* ]]
The *getsockname*(2) system call.

getsockopt *fildes level name data data_size*
The *getsockopt*(2) system call.

gettimeofday [ *tv* [ *tz* ] ]
The *gettimeofday*(2) system call.

getw *fp* The *getw*(3) system call.

ioctl *fildes request data*
The *ioctl*(2) system call.

kill *pid sig*
The *kill*(2) system call.

lchmod *pathname mode*
The *lchmod*(2) system call.

lchown *pathname owner group*
The *lchown*(2) system call.

link *oldpath newpath*
The *link*(2) system call.

listen *fildes backlog*
The *listen*(2) system call.

lseek *fildes offset whence*
The *lseek*(2) system call.

lstat *pathname*
The *lstat*(2) system call.

malloc *size*
The *malloc*(3) system call.

mkdir *pathname* [ *mode* ]
The *mkdir*(2) system call.

mkdtemp *pathname*
The *mkdtemp*(3) system call.

mknod *pathname mode dev*
The *mknod*(2) system call.

mkostemp *templat flags*
The *mkostemp*(3) system call.

mkstemp *templat*
The *mkstemp*(3) system call.

mktemp *pathname*
The *mktemp*(3) system call.

mmap *data data_size prot flags fildes offset*
The *mmap*(2) system call.

munmap *data data_size*
The *munmap*(2) system call.

nice *inc* The *nice*(2) system call.

open *pathname flags* [ *mode* ]
The *open*(2) system call. The *pathname* argument may need to be quoted to insulate white space and punctuation from the shell. The *flags* argument may be numeric or symbolic. The *mode* argument may be numeric or symbolic.

opendir *pathname*
The *opendir*(3) system call.

pathconf *pathname name*
The *pathconf*(3) system call.

pclose *fp*
The *pclose*(3) system call.

pipe *pipefd*
The *pipe*(2) system call.

poll *fds nfds timeout*
The *poll*(2) system call.

popen *command flags*
The *popen*(3) system call.

pread *fildes data data_size offset*
The *pread*(2) system call.

ptrace *request pid addr data*
The *ptrace*(2) system call.

putc *c fp* The *putc*(3) system call.

putchar *c*
The *putchar*(3) system call.

putenv *string*
The *putenv*(3) system call.

puts *s* The *puts*(3) system call.

putw *value fp*
The *putw*(3) system call.

pwrite *fildes data data_size offset*
The *pwrite*(2) system call.

raise *sig* The *raise*(3) system call.

read *fildes data data-size*
The *read*(2) system call.

realloc *ptr size*
The *realloc*(3) system call.

realpath *pathname resolved_pathname*
The *realpath*(3) system call.

rename *oldpath newpath*
The *rename*(2) system call.

readv *fildes iov* ...
The *readv*(2) system call.

select *nfds readfds writefds exceptfds timeout*
The *select*(2) system call.

setbuf *fp data*
The *setbuf*(3) system call.

setbuffer *fp data size*
The *setbuffer*(3) system call.

setdomainname *data data_size*
The *setdomainname*(2) system call.

setenv *name value overwrite*
The *setenv*(3) system call.

setgid *gid*
The *setgid*(2) system call.

setgroups *data_size data*
The *setgroups*(2) system call.

sethostname *name* [ *name_size* ]
The *sethostname*(2) system call.

setlinebuf *fp*
The *setlinebuf*(3) system call.

setpgid [ *pid* [ *pgid* ]]
The *setpgid*(2) system call.

setpgrp *pid pgid*
The *setpgrp*(2) system call.

setregid *rgid egid*
The *setregid*(2) system call.

setreuid *ruid euid*
The *setreuid*(2) system call.

setresgid *rgid egid sgid*
The *setresgid*(2) system call.

setresuid *ruid euid suid*
The *setresuid*(2) system call.

setreuid *ruid euid*
The *setreuid*(2) system call.

setsid The *setsid*(2) system call.

setsockopt *fildes level name data data_size*
The *setsockopt*(2) system call.

setuid *uid*
The *setuid*(2) system call.

setvbuf *fp data mode size*
The *setvbuf*(3) system call.

shmat *shmid shmaddr shmflg*
The *shmat*(2) system call.

shmctl *shmid command data*
The *shmctl*(2) system call.

signalfd *fildes mask flags*
The *signalfd*(2) system call.

socket *domain type protocol*
The *socket*(2) system call.

socketpair *domain type protocol sv*
The *socketpair*(2) system call.

stat *pathname*
The *stat*(2) system call.

statfs *pathname data*
The *statfs*(2) system call.

statvfs *pathname data*
The *statvfs*(2) system call.

stime *t* The *stime*(2) system call.

strdup *data*
The *strdup*(3) system call.

strerror The error given will be printed out with all known detail.

strndup *data data_size*
The *strndup*(3) system call.

strtod *nptr endptr*
The *strtod*(3) system call.

strtof *nptr endptr*
The *strtof*(3) system call.

strtol *nptr endptr base*
The *strtol*(3) system call.

strtold *nptr endptr*
The *strtold*(3) system call.

strtoll *nptr endptr base*
The *strtoll*(3) system call.

strtoul *nptr endptr base*
The *strtoul*(3) system call.

strtoull *nptr endptr base*
The *strtoull*(3) system call.

symlink *oldpath newpath*
The *symlink*(2) system call.

system *command*
The *system*(3) system call.

tcdrain *fildes*
The *tcdrain*(3) system call.

tcflow *fildes action*
The *tcflow*(3) system call.

tcflush *fildes selector*
The *tcflush*(3) system call.

tcgetattr *fildes data*
The *tcgetattr*(3) system call.

tcsendbreak *fildes duration*
The *tcsendbreak*(3) system call.

tcsetattr *fildes options data*
The *tcsetattr*(3) system call.

telldir *dir*
The *telldir*(3) system call.

tempnam *dir prefix*
The *tempnam*(3) system call.

time *t* The *time*(2) system call.

timerfd_create *clockid flags*
The *timerfd_create*(2) system call.

tmpfile The *tmpfile*(3) system call.

tmpnam *pathname*
The *tmpnam*(3) system call.

truncate *pathname size*
The *truncate*(2) system call.

ungetc *c fp*
The *ungetc*(3) system call.

unlink *pathname*
The *unlink*(2) system call.

unsetenv *name*
The *unsetenv*(3) system call.

ustat *dev ubuf*
The *ustat*(2) system call.

utime *pathname* [ *times* ]
The *utime*(2) system call.

utimens *pathname* [ *data* ]
The *utimens*(2) system call.

utimensat [ *fildes* ] *pathname* [ *data* [ *flags* ]]
The *utimensat*(2) system call.

utimes *pathname data*
The *utimes*(2) system call.

vfork The *vfork*(2) system call.

wait *status*
The *wait*(2) system call.

wait3 *status options rusage*
The *wait3*(2) system call.

wait4 *pid status options rusage*
The *wait4*(2) system call.

waitpid *pid status options*
: The *waitpid*(2) system call.

write *fildes data data-size*
: The *write*(2) system call.

writev *fildes data data-size*
: The *writev*(2) system call.

Do not include the perentheses used to make the call.

## OPTIONS

The explain command understands the following options:

**−E**
: The exit staus, success or fail, will be printed immediately before the *access* command terminates.

**−e** *number*
: The value of *errno* as a number (*e.g.* 2), or as a symbol (*e.g.* ENOENT), or as the text of its meaning (*e.g.* No such file or directory*). You will need quotes to insulate spaces and punctuation from the shell.*

**−V**
: Print the version of the *explain* executing.

## EXIT STATUS

The explain command exits with status 1 on any error. The explain command only exits with status 0 if there are no errors.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_lca2010 − No medium found: when it's time to stop trying to read *strerror*(3)'s mind.

## MOTIVATION

The idea for libexplain occurred to me back in the early 1980s. Whenever a system call returns an error, the kernel knows exactly what went wrong... and compresses this into less that 8 bits of *errno*. User space has access to the same data as the kernel, it should be possible for user space to figure out exactly what happened to provoke the error return, and use this to write good error messages.

Could it be that simple?

### Error messages as finesse

Good error messages are often those "one percent" tasks that get dropped when schedule pressure squeezes your project. However, a good error message can make a huge, disproportionate improvement to the user experience, when the user wanders into scarey unknown territory not usually encountered. This is no easy task.

As a larval programmer, the author didn't see the problem with (completely accurate) error messages like this one:

```
floating exception (core dumped)
```

until the alternative non-programmer interpretation was pointed out. But that isn't the only thing wrong with Unix error messages. How often do you see error messages like:

```
$ ./stupid
can't open file
$
```

There are two options for a developer at this point:

1. you can run a debugger, such as *gdb*(1), or
2. you can use *strace*(1) or *truss*(1) to look inside.

- Remember that your users may not even have access to these tools, let alone the ability to use them. (It's a very long time since *Unix beginner* meant "has only written *one* device driver".)

In this example, however, using *strace*(1) reveals

```
$ strace -e trace=open ./stupid
open("some/file", O_RDONLY) = -1 ENOENT (No such file or directory)
can't open file
$
```

This is considerably more information than the error message provides. Typically, the stupid source code looks like this

```
int fd = open("some/thing", O_RDONLY);
if (fd < 0)
{
    fprintf(stderr, "can't open file\n");
    exit(1);
}
```

The user isn't told *which* file, and also fails to tell the user *which* error. Was the file even there? Was there a permissions problem? It does tell you it was trying to open a file, but that was probably by accident.

Grab your clue stick and go beat the larval programmer with it. Tell him about *perror*(3). The next time you use the program you see a different error message:

```
$ ./stupid
open: No such file or directory
$
```

Progress, but not what we expected. How can the user fix the problem if the error message doesn't tell him

what the problem was? Looking at the source, we see

```
int fd = open("some/thing", O_RDONLY);
if (fd < 0)
{
    perror("open");
    exit(1);
}
```

Time for another run with the clue stick. This time, the error message takes one step forward and one step back:

```
$ ./stupid
some/thing: No such file or directory
$
```

Now we know the file it was trying to open, but are no longer informed that it was *open*(2) that failed. In this case it is probably not significant, but it can be significant for other system calls. It could have been *creat*(2) instead, an operation implying that different permissions are necessary.

```
const char *filename = "some/thing";
int fd = open(filename, O_RDONLY);
if (fd < 0)
{
    perror(filename);
    exit(1);
}
```

The above example code is unfortunately typical of non-larval programmers as well. Time to tell our padawan learner about the *strerror*(3) system call.

```
$ ./stupid
open some/thing: No such file or directory
$
```

This maximizes the information that can be presented to the user. The code looks like this:

```
const char *filename = "some/thing";
int fd = open(filename, O_RDONLY);
if (fd < 0)
{
    fprintf(stderr, "open %s: %s\n", filename, strerror(errno));
    exit(1);
}
```

Now we have the system call, the filename, and the error string. This contains all the information that *strace*(1) printed. That's as good as it gets.

Or is it?

### Limitations of `perror` and `strerror`

The problem the author saw, back in the 1980s, was that the error message is incomplete. Does "no such file or directory" refer to the "*some*" directory, or to the "*thing*" file in the "*some*" directory?

A quick look at the man page for *strerror*(3) is telling:

> strerror − return string describing error number

Note well: it is describing the error *number*, not the error.

On the other hand, the kernel *knows* what the error was. There was a specific point in the kernel code, caused by a specific condition, where the kernel code branched and said "no". Could a user-space program figure out the specific condition and write a better error message?

However, the problem goes deeper. What if the problem occurs during the *read*(2) system call, rather than the *open*(2) call? It is simple for the error message associated with *open*(2) to include the file name, it's

right there. But to be able to include a file name in the error associated with the *read*(2) system call, you have to pass the file name all the way down the call stack, as well as the file descriptor.

And here is the bit that grates: the kernel already knows what file name the file descriptor is associated with. Why should a programmer have to pass redundant data all the way down the call stack just to improve an error message that may never be issued? In reality, many programmers don't bother, and the resulting error messages are the worse for it.

But that was the 1980s, on a PDP11, with limited resources and no shared libraries. Back then, no flavor of Unix included `/proc` even in rudimentary form, and the *lsof*(1) program was over a decade away. So the idea was shelved as impractical.

### Level Infinity Support

Imagine that you are level infinity support. Your job description says that you never *ever* have to talk to users. Why, then, is there still a constant stream of people wanting you, the local Unix guru, to decipher yet another error message?

Strangely, 25 years later, despite a simple permissions system, implemented with complete consistency, most Unix users still have no idea how to decode "No such file or directory", or any of the other cryptic error messages they see every day. Or, at least, cryptic to them.

Wouldn't it be nice if first level tech support didn't need error messages deciphered? Wouldn't it be nice to have error messages that users could understand without calling tech support?

These days `/proc` on Linux is more than able to provide the information necessary to decode the vast majority of error messages, and point the user to the proximate cause of their problem. On systems with a limited `/proc` implementation, the *lsof*(1) command can fill in many of the gaps.

In 2008, the stream of translation requests happened to the author way too often. It was time to re-examine that 25 year old idea, and libexplain is the result.

## USING THE LIBRARY

The interface to the library tries to be consistent, where possible. Let's start with an example using *strerror*(3):

```
if (rename(old_path, new_path) < 0)
{
    fprintf(stderr, "rename %s %s: %s\n", old_path, new_path,
        strerror(errno));
    exit(1);
}
```

The idea behind libexplain is to provide a *strerror*(3) equivalent for **each** system call, tailored specifically to that system call, so that it can provide a more detailed error message, containing much of the information you see under the "ERRORS" heading of section 2 and 3 *man* pages, supplemented with information about actual conditions, actual argument values, and system limits.

### The Simple Case

The *strerror*(3) replacement:

```
if (rename(old_path, new_path) < 0)
{
    fprintf(stderr, "%s\n", explain_rename(old_path, new_path));
    exit(1);
}
```

### The Errno Case

It is also possible to pass an explicit *errno*(3) value, if you must first do some processing that would disturb *errno*, such as error recovery:

```
if (rename(old_path, new_path < 0))
{
    int old_errno = errno;
```

```
        ...code that disturbs errno...
        fprintf(stderr, "%s\n", explain_errno_rename(old_errno,
            old_path, new_path));
        exit(1);
    }
```

**The Multi-thread Cases**

Some applications are multi-threaded, and thus are unable to share libexplain's internal buffer. You can supply your own buffer using

```
    if (unlink(pathname))
    {
        char message[3000];
        explain_message_unlink(message, sizeof(message), pathname);
        error_dialog(message);
        return -1;
    }
```

And for completeness, both *errno*(3) and thread-safe:

```
    ssize_t nbytes = read(fd, data, sizeof(data));
    if (nbytes < 0)
    {
        char message[3000];
        int old_errno = errno;
        ...error recovery...
        explain_message_errno_read(message, sizeof(message),
            old_errno, fd, data, sizeof(data));
        error_dialog(message);
        return -1;
    }
```

These are replacements for *strerror_r*(3), on systems that have it.

**Interface Sugar**

A set of functions added as convenience functions, to woo programmers to use the libexplain library, turn out to be the author's most commonly used libexplain functions in command line programs:

```
    int fd = explain_creat_or_die(filename, 0666);
```

This function attempts to create a new file. If it can't, it prints an error message and exits with EXIT_FAILURE. If there is no error, it returns the new file descriptor.

A related function:

```
    int fd = explain_creat_on_error(filename, 0666);
```

will print the error message on failure, but also returns the original error result, and *errno*(3) is unmolested, as well.

**All the other system calls**

In general, every system call has its own include file

```
    #include <libexplain/name.h>
```

that defines function prototypes for six functions:

- `explain_`*name*,
- `explain_errno_`*name*,
- `explain_message_`*name*,
- `explain_message_errno_`*name*,

- `explain_`*name*`_or_die` and
- `explain_`*name*`_on_error`.

Every function prototype has Doxygen documentation, and this documentation *is not* stripped when the include files are installed.

The *wait*(2) system call (and friends) have some extra variants that also interpret failure to be an exit status that isn't EXIT_SUCCESS. This applies to *system*(3) and *pclose*(3) as well.

Coverage includes 187 system calls and 547 ioctl requests. There are many more system calls yet to implement. System calls that never return, such as *exit*(2), are not present in the library, and will never be. The *exec* family of system calls *are* supported, because they return when there is an error.

**Cat**

This is what a hypothetical "cat" program could look like, with full error reporting, using libexplain.

```
#include <libexplain/libexplain.h>
#include <stdlib.h>
#include <unistd.h>
```

There is one include for libexplain, plus the usual suspects. (If you wish to reduce the preprocessor load, you can use the specific `<libexplain/`*name*`.h>` includes.)

```
static void
process(FILE *fp)
{
    for (;;)
    {
        char buffer[4096];
        size_t n = explain_fread_or_die(buffer, 1, sizeof(buffer), fp);
        if (!n)
            break;
        explain_fwrite_or_die(buffer, 1, n, stdout);
    }
}
```

The *process* function copies a file stream to the standard output. Should an error occur for either reading or writing, it is reported (and the pathname will be included in the error) and the command exits with EXIT_FAILURE. We don't even worry about tracking the pathnames, or passing them down the call stack.

```
int
main(int argc, char **argv)
{
    for (;;)
    {
        int c = getopt(argc, argv, "o:");
        if (c == EOF)
            break;
        switch (c)
        {
        case 'o':
            explain_freopen_or_die(optarg, "w", stdout);
            break;
```

The fun part of this code is that libexplain can report errors *including the pathname* even if you **don't** explicitly re-open stdout as is done here. We don't even worry about tracking the file name.

```
        default:
            fprintf(stderr, "Usage: %ss [ -o <filename> ] <filename>...\n",
                argv[0]);
            return EXIT_FAILURE;
```

```
            }
        }
        if (optind == argc)
            process(stdin);
        else
        {
            while (optind < argc)
            {
                FILE *fp = explain_fopen_or_die(argv[optind]++, "r");
                process(fp);
                explain_fclose_or_die(fp);
            }
        }
```

The standard output will be closed implicitly, but too late for an error report to be issued, so we do that here, just in case the buffered I/O hasn't written anything yet, and there is an ENOSPC error or something.

```
        explain_fflush_or_die(stdout);
        return EXIT_SUCCESS;
    }
```

That's all. Full error reporting, clear code.

**Rusty's Scale of Interface Goodness**

For those of you not familiar with it, Rusty Russel's "How Do I Make This Hard to Misuse?" page is a must-read for API designers.
http://ozlabs.org/~rusty/index.cgi/tech/2008-03-30.html

*10. It's impossible to get wrong.*

Goals need to be set high, ambitiously high, lest you accomplish them and think you are finished when you are not.

The libexplain library detects bogus pointers and many other bogus system call parameters, and generally tries to avoid segfaults in even the most trying circumstances.

The libexplain library is designed to be thread safe. More real-world use will likely reveal places this can be improved.

The biggest problem is with the actual function names themselves. Because C does not have name-spaces, the libexplain library always uses an `explain_` name prefix. This is the traditional way of creating a pseudo-name-space in order to avoid symbol conflicts. However, it results in some unnatural-sounding names.

*9. The compiler or linker won't let you get it wrong.*

A common mistake is to use `explain_open` where `explain_open_or_die` was intended. Fortunately, the compiler will often issue a type error at this point (*e.g.* can't assign `const char *` rvalue to an `int` lvalue).

*8. The compiler will warn if you get it wrong.*

If `explain_rename` is used when `explain_rename_or_die` was intended, this can cause other problems. GCC has a useful `warn_unused_result` function attribute, and the libexplain library attaches it to all the `explain_`*name* function calls to produce a warning when you make this mistake. Combine this with *gcc −Werror* to promote this to level 9 goodness.

*7. The obvious use is (probably) the correct one.*

The function names have been chosen to convey their meaning, but this is not always successful. While `explain_`*name*`_or_die` and `explain_`*name*`_on_error` are fairly descriptive, the less-used thread safe variants are harder to decode. The function prototypes help the compiler towards understanding, and the Doxygen comments in the header files help the user towards understanding.

*6. The name tells you how to use it.*

It is particularly important to read `explain_`*name*`_or_die` as "explain (*name* or die)". Using a consistent `explain_` name-space prefix has some unfortunate side-effects in the obviousness department, as well.

The order of words in the names also indicate the order of the arguments. The argument lists always *end* with the same arguments as passed to the system call; *all of them*. If `_errno_` appears in the name, its argument always precedes the system call arguments. If `_message_` appears in the name, its two arguments always come first.

*5. Do it right or it will break at runtime.*

The libexplain library detects bogus pointers and many other bogus system call parameters, and generally tries to avoid segfaults in even the most trying circumstances. It should never break at runtime, but more real-world use will no doubt improve this.

Some error messages are aimed at developers and maintainers rather than end users, as this can assist with bug resolution. Not so much "break at runtime" as "be informative at runtime" (after the system call barfs).

*4. Follow common convention and you'll get it right.*

Because C does not have name-spaces, the libexplain library always uses an `explain_` name prefix. This is the traditional way of creating a pseudo-name-space in order to avoid symbol conflicts.

The trailing arguments of all the libexplain call are identical to the system call they are describing. This is intended to provide a consistent convention in common with the system calls themselves.

*3. Read the documentation and you'll get it right.*

The libexplain library aims to have complete Doxygen documentation for each and every public API call (and internally as well).

## MESSAGE CONTENT

Working on libexplain is a bit like looking at the underside of your car when it is up on the hoist at the mechanic's. There's some ugly stuff under there, plus mud and crud, and users rarely see it. A good error message needs to be informative, even for a user who has been fortunate enough not to have to look at the under-side very often, and also informative for the mechanic listening to the user's description over the phone. This is no easy task.

Revisiting our first example, the code would like this if it uses libexplain:

```
int fd = explain_open_or_die("some/thing", O_RDONLY, 0);
```

will fail with an error message like this

```
open(pathname = "some/file", flags = O_RDONLY) failed, No such
file or directory (2, ENOENT) because there is no "some" directory
in the current directory
```

This breaks down into three pieces

*system-call* `failed,` *system-error* `because` *explanation*

### Before Because

It is possible to see the part of the message before "because" as overly technical to non-technical users, mostly as a result of accurately printing the system call itself at the beginning of the error message. And it looks like *strace*(1) output, for bonus geek points.

```
open(pathname = "some/file", flags = O_RDONLY) failed, No such
file or directory (2, ENOENT)
```

This part of the error message is essential to the developer when he is writing the code, and equally important to the maintainer who has to read bug reports and fix bugs in the code. It says exactly what failed.

If this text is not presented to the user then the user cannot copy-and-paste it into a bug report, and if it isn't in the bug report the maintainer can't know what actually went wrong.

Frequently tech staff will use *strace*(1) or *truss*(1) to get this exact information, but this avenue is not open when reading bug reports. The bug reporter's system is far far away, and, by now, in a far different state. Thus, this information needs to be in the bug report, which means it must be in the error message.

The system call representation also gives context to the rest of the message. If need arises, the offending system call argument may be referred to by name in the explanation after "because". In addition, all strings are fully quoted and escaped C strings, so embedded newlines and non-printing characters will not cause the user's terminal to go haywire.

The *system-error* is what comes out of *strerror*(2), plus the error symbol. Impatient and expert sysadmins could stop reading at this point, but the author's experience to date is that reading further is rewarding. (If it isn't rewarding, it's probably an area of libexplain that can be improved. Code contributions are welcome, of course.)

**After Because**

This is the portion of the error message aimed at non-technical users. It looks beyond the simple system call arguments, and looks for something more specific.

```
there is no "some" directory in the current directory
```

This portion attempts to explain the proximal cause of the error in plain language, and it is here that internationalization is essential.

In general, the policy is to include as much information as possible, so that the user doesn't need to go looking for it (and doesn't leave it out of the bug report).

**Internationalization**

Most of the error messages in the libexplain library have been internationalized. There are no localizations as yet, so if you want the explanations in your native language, please contribute.

The "most of" qualifier, above, relates to the fact that the proof-of-concept implementation did not include internationalization support. The code base is being revised progressively, usually as a result of refactoring messages so that each error message string appears in the code exactly once.

Provision has been made for languages that need to assemble the portions of

*system-call* `failed,` *system-error* `because` *explanation*

in different orders for correct grammar in localized error messages.

**Postmortem**

There are times when a program has yet to use libexplain, and you can't use *strace*(1) either. There is an *explain*(1) command included with libexplain that can be used to decipher error messages, if the state of the underlying system hasn't changed too much.

```
$ explain rename foo /tmp/bar/baz -e ENOENT
rename(oldpath = "foo", newpath = "/tmp/bar/baz") failed, No such
file or directory (2, ENOENT) because there is no "bar" directory
in the newpath "/tmp" directory
$
```

Note how the path ambiguity is resolved by using the system call argument name. Of course, you have to know the error and the system call for *explain*(1) to be useful. As an aside, this is one of the ways used by the libexplain automatic test suite to verify that libexplain is working.

**Philosophy**

"Tell me everything, including stuff I didn't know to look for."

The library is implemented in such a way that when statically linked, only the code you actually use will be linked. This is achieved by having one function per source file, whenever feasible.

When it is possible to supply more information, libexplain will do so. The less the user has to track down for themselves, the better. This means that UIDs are accompanied by the user name, GIDs are

accompanied by the group name, PIDs are accompanied by the process name, file descriptors and streams are accompanied by the pathname, *etc*.

When resolving paths, if a path component does not exist, libexplain will look for similar names, in order to suggest alternatives for typographical errors.

The libexplain library tries to use as little heap as possible, and usually none. This is to avoid perturbing the process state, as far as possible, although sometimes it is unavoidable.

The libexplain library attempts to be thread safe, by avoiding global variables, keeping state on the stack as much as possible. There is a single common message buffer, and the functions that use it are documented as not being thread safe.

The libexplain library does not disturb a process's signal handlers. This makes determining whether a pointer would segfault a challenge, but not impossible.

When information is available via a system call as well as available through a `/proc` entry, the system call is preferred. This is to avoid disturbing the process's state. There are also times when no file descriptors are available.

The libexplain library is compiled with large file support. There is no large/small schizophrenia. Where this affects the argument types in the API, and error will be issued if the necessary large file defines are absent.

FIXME: Work is needed to make sure that file system quotas are handled in the code. This applies to some *getrlimit*(2) boundaries, as well.

There are cases when relatives paths are uninformative. For example: system daemons, servers and background processes. In these cases, absolute paths are used in the error explanations.

## PATH RESOLUTION

Short version: see *path_resolution*(7).

Long version: Most users have never heard of *path_resolution*(7), and many advanced users have never read it. Here is an annotated version:

### Step 1: Start of the resolution process

If the pathname starts with the slash ("/") character, the starting lookup directory is the root directory of the calling process.

If the pathname does not start with the slash("/") character, the starting lookup directory of the resolution process is the current working directory of the process.

### Step 2: Walk along the path

Set the current lookup directory to the starting lookup directory. Now, for each non-final component of the pathname, where a component is a substring delimited by slash ("/") characters, this component is looked up in the current lookup directory.

If the process does not have search permission on the current lookup directory, an EACCES error is returned ("Permission denied").

```
open(pathname = "/home/archives/.ssh/private_key", flags =
O_RDONLY) failed, Permission denied (13, EACCES) because the
process does not have search permission to the pathname
"/home/archives/.ssh" directory, the process effective GID 1000
"pmiller" does not match the directory owner 1001 "archives" so
the owner permission mode "rwx" is ignored, the others permission
mode is "---", and the process is not privileged (does not have
the DAC_READ_SEARCH capability)
```

If the component is not found, an ENOENT error is returned ("No such file or directory").

```
unlink(pathname = "/home/microsoft/rubbish") failed, No such file
or directory (2, ENOENT) because there is no "microsoft" directory
in the pathname "/home" directory
```

There is also some support for users when they mis-type pathnames, making suggestions when ENOENT is returned:

```
open(pathname = "/user/include/fcntl.h", flags = O_RDONLY) failed,
No such file or directory (2, ENOENT) because there is no "user"
directory in the pathname "/" directory, did you mean the "usr"
directory instead?
```

If the component is found, but is neither a directory nor a symbolic link, an ENOTDIR error is returned ("Not a directory").

```
open(pathname = "/home/pmiller/.netrc/lca", flags = O_RDONLY)
failed, Not a directory (20, ENOTDIR) because the ".netrc" regular
file in the pathname "/home/pmiller" directory is being used as a
directory when it is not
```

If the component is found and is a directory, we set the current lookup directory to that directory, and go to the next component.

If the component is found and is a symbolic link (symlink), we first resolve this symbolic link (with the current lookup directory as starting lookup directory). Upon error, that error is returned. If the result is not a directory, an ENOTDIR error is returned.

```
unlink(pathname = "/tmp/dangling/rubbish") failed, No such file or
directory (2, ENOENT) because the "dangling" symbolic link in the
pathname "/tmp" directory refers to "nowhere" that does not exist
```

If the resolution of the symlink is successful and returns a directory, we set the current lookup directory to that directory, and go to the next component. Note that the resolution process here involves recursion. In order to protect the kernel against stack overflow, and also to protect against denial of service, there are limits on the maximum recursion depth, and on the maximum number of symbolic links followed. An ELOOP error is returned when the maximum is exceeded ("Too many levels of symbolic links").

```
open(pathname = "/tmp/dangling", flags = O_RDONLY) failed, Too
many levels of symbolic links (40, ELOOP) because a symbolic link
loop was encountered in pathname, starting at "/tmp/dangling"
```

It is also possible to get an ELOOP or EMLINK error if there are too many symlinks, but no loop was detected.

```
open(pathname = "/tmp/rabbit-hole", flags = O_RDONLY) failed, Too
many levels of symbolic links (40, ELOOP) because too many
symbolic links were encountered in pathname (8)
```

Notice how the actual limit is also printed.

**Step 3: Find the final entry**

The lookup of the final component of the pathname goes just like that of all other components, as described in the previous step, with two differences:

(i) The final component need not be a directory (at least as far as the path resolution process is concerned. It may have to be a directory, or a non-directory, because of the requirements of the specific system call).

(ii) It is not necessarily an error if the final component is not found; maybe we are just creating it. The details on the treatment of the final entry are described in the manual pages of the specific system calls.

(iii) It is also possible to have a problem with the last component if it is a symbolic link and it should not be followed. For example, using the *open*(2) O_NOFOLLOW flag:

```
open(pathname = "a-symlink", flags = O_RDONLY | O_NOFOLLOW) failed,
Too many levels of symbolic links (ELOOP) because O_NOFOLLOW was
specified but pathname refers to a symbolic link
```

(iv) It is common for users to make mistakes when typing pathnames. The libexplain library attempts to make suggestions when ENOENT is returned, for example:

```
open(pathname = "/usr/include/filecontrl.h", flags = O_RDONLY)
failed, No such file or directory (2, ENOENT) because there is no
"filecontrl.h" regular file in the pathname "/usr/include"
directory, did you mean the "fcntl.h" regular file instead?
```

(v) It is also possible that the final component is required to be something other than a regular file:

```
readlink(pathname = "just-a-file", data = 0x7F930A50, data_size =
4097) failed, Invalid argument (22, EINVAL) because pathname is a
regular file, not a symbolic link
```

(vi) FIXME: handling of the "t" bit.

**Limits**

There are a number of limits with regards to pathnames and filenames.

Pathname length limit

There is a maximum length for pathnames. If the pathname (or some intermediate pathname obtained while resolving symbolic links) is too long, an ENAMETOOLONG error is returned ("File name too long"). Notice how the system limit is included in the error message.

```
open(pathname = "very...long", flags = O_RDONLY) failed, File name
too long (36, ENAMETOOLONG) because pathname exceeds the system
maximum path length (4096)
```

Filename length limit

Some Unix variants have a limit on the number of bytes in each path component. Some of them deal with this silently, and some give ENAMETOOLONG; the libexplain library uses *pathconf*(3) _PC_NO_TRUNC to tell which. If this error happens, the libexplain library will state the limit in the error message, the limit is obtained from *pathconf*(3) _PC_NAME_MAX. Notice how the system limit is included in the error message.

```
open(pathname = "system7/only-had-14-characters", flags = O_RDONLY)
failed, File name too long (36, ENAMETOOLONG) because
"only-had-14-characters" component is longer than the system
limit (14)
```

Empty pathname

In the original Unix, the empty pathname referred to the current directory. Nowadays POSIX decrees that an empty pathname must not be resolved successfully.

```
open(pathname = "", flags = O_RDONLY) failed, No such file or
directory (2, ENOENT) because POSIX decrees that an empty
pathname must not be resolved successfully
```

**Permissions**

The permission bits of a file consist of three groups of three bits. The first group of three is used when the effective user ID of the calling process equals the owner ID of the file. The second group of three is used when the group ID of the file either equals the effective group ID of the calling process, or is one of the supplementary group IDs of the calling process. When neither holds, the third group is used.

```
open(pathname = "/etc/passwd", flags = O_WRONLY) failed,
Permission denied (13, EACCES) because the process does not have
write permission to the "passwd" regular file in the pathname
"/etc" directory, the process effective UID 1000 "pmiller" does
not match the regular file owner 0 "root" so the owner permission
mode "rw-" is ignored, the others permission mode is "r--", and
the process is not privileged (does not have the DAC_OVERRIDE
```

```
capability)
```

Some considerable space is given to this explanation, as most users do not know that this is how the permissions system works. In particular: the owner, group and other permissions are exclusive, they are not "OR"ed together.

## STRANGE AND INTERESTING SYSTEM CALLS

The process of writing a specific error handler for each system call often reveals interesting quirks and boundary conditions, or obscure *errno*(3) values.

### ENOMEDIUM, No medium found

The act of copying a CD was the source of the title for this paper.

```
$ dd if=/dev/cdrom of=fubar.iso
dd: opening "/dev/cdrom": No medium found
$
```

The author wondered why his computer was telling him there is no such thing as a psychic medium. Quite apart from the fact that huge numbers of native English speakers are not even aware that "media" is a plural, let alone that "medium" is its singular, the string returned by *strerror*(3) for ENOMEDIUM is so terse as to be almost completely free of content.

When *open*(2) returns ENOMEDIUM it would be nice if the libexplain library could expand a little on this, based on the type of drive it is. For example:

... because there is no disk in the floppy drive
... because there is no disc in the CD-ROM drive
... because there is no tape in the tape drive
... because there is no memory stick in the card reader

And so it came to pass...

```
open(pathname = "/dev/cdrom", flags = O_RDONLY) failed, No medium
found (123, ENOMEDIUM) because there does not appear to be a disc
in the CD-ROM drive
```

The trick, that the author was previously unaware of, was to open the device using the O_NONBLOCK flag, which will allow you to open a drive with no medium in it. You then issue device specific *ioctl*(2) requests until you figure out what the heck it is. (Not sure if this is POSIX, but it also seems to work that way in BSD and Solaris, according to the *wodim*(1) sources.)

Note also the differing uses of "disk" and "disc" in context. The CD standard originated in France, but everything else has a "k".

### EFAULT, Bad address

Any system call that takes a pointer argument can return EFAULT. The libexplain library can figure out which argument is at fault, and it does it without disturbing the process (or thread) signal handling.

When available, the *mincore*(2) system call is used, to ask if the memory region is valid. It can return three results: mapped but not in physical memory, mapped and in physical memory, and not mapped. When testing the validity of a pointer, the first two are "yes" and the last one is "no".

Checking C strings are more difficult, because instead of a pointer and a size, we only have a pointer. To determine the size we would have to find the NUL, and that could segfault, catch-22.

To work around this, the libexplain library uses the *lstat*(2) sysem call (with a known good second argument) to test C strings for validity. A failure return && errno == EFAULT is a "no", and anythng else is a "yes". This, of course limits strings to PATH_MAX characters, but that usually isn't a problem for the libexplain library, because that is almost always the longest strings it cares about.

### EMFILE, Too many open files

This error occurs when a process already has the maximum number of file descriptors open. If the actual limit is to be printed, and the libexplain library tries to, you can't open a file in `/proc` to read what it is.

```
open_max = sysconf(_SC_OPEN_MAX);
```

This one wan't so difficult, there is a *sysconf*(3) way of obtaining the limit.

**ENFILE, Too many open files in system**

This error occurs when the system limit on the total number of open files has been reached. In this case there is no handy *sysconf*(3) way of obtain the limit.

Digging deeper, one may discover that on Linux there is a `/proc` entry we could read to obtain this value. Catch-22: we are out of file descriptors, so we can't open a file to read the limit.

On Linux there is a system call to obtain it, but it has no [e]glibc wrapper function, so you have to all it very carefully:

```
long
explain_maxfile(void)
{
#ifdef __linux__
    struct __sysctl_args args;
    int32_t maxfile;
    size_t maxfile_size = sizeof(maxfile);
    int name[] = { CTL_FS, FS_MAXFILE };
    memset(&args, 0, sizeof(struct __sysctl_args));
    args.name = name;
    args.nlen = 2;
    args.oldval = &maxfile;
    args.oldlenp = &maxfile_size;
    if (syscall(SYS__sysctl, &args) >= 0)
        return maxfile;
#endif
    return -1;
}
```

This permits the limit to be included in the error message, when available.

**EINVAL "Invalid argument"** *vs* **ENOSYS "Function not implemented"**

Unsupported actions (such as *symlink*(2) on a FAT file system) are not reported consistently from one system call to the next. It is possible to have either EINVAL or ENOSYS returned.

As a result, attention must be paid to these error cases to get them right, particularly as the EINVAL could also be referring to problems with one or more system call arguments.

**Note that *errno*(3) is not always set**

There are times when it is necessary to read the [e]glibc sources to determine how and when errors are returned for some system calls.

*feof*(3), *fileno*(3)
: It is often assumed that these functions cannot return an error. This is only true if the *stream* argument is valid, however they are capable of detecting an invalid pointer.

*fpathconf*(3), *pathconf*(3)
: The return value of *fpathconf*(2) and *pathconf*(2) could legitimately be −1, so it is necessary to see if *errno*(3) has been explicitly set.

*ioctl*(2)
: The return value of *ioctl*(2) could legitimately be −1, so it is necessary to see if *errno*(3) has been explicitly set.

*readdir*(3)
: The return value of *readdir*(3) is NULL for both errors and end-of-file. It is necessary to see if *errno*(3) has been explicitly set.

*setbuf*(3), *setbuffer*(3), *setlinebuf*(3), *setvbuf*(3)
All but the last of these functions return void. And *setvbuf*(3) is only documented as returning "non-zero" on error. It is necessary to see if *errno*(3) has been explicitly set.

*strtod*(3), *strtol*(3), *strtold*(3), *strtoll*(3), *strtoul*(3), *strtoull*(3)
These functions return 0 on error, but that is also a legitimate return value. It is necessary to see if *errno*(3) has been explicitly set.

*ungetc*(3)
While only a single character of backup is mandated by the ANSI C standard, it turns out that [e]glibc permits more... but that means it can fail with ENOMEM. It can also fail with EBADF if *fp* is bogus. Most difficult of all, if you pass EOF an error return occurs, but errno is not set.

The libexplain library detects all of these errors correctly, even in cases where the error values are poorly documented, if at all.

**ENOSPC, No space left on device**

When this error refers to a file on a file system, the libexplain library prints the mount point of the file system with the problem. This can make the source of the error much clearer.

```
write(fildes = 1 "example", data = 0xbfff2340, data_size = 5)
failed, No space left on device (28, ENOSPC) because the file
system containing fildes ("/home") has no more space for data
```

As more special device support is added, error messages are expected to include the device name and actual size of the device.

**EROFS, Read-only file system**

When this error refers to a file on a file system, the libexplain library prints the mount point of the file system with the problem. This can make the source of the error much clearer.

As more special device support is added, error messages are expected to include the device name and type.

```
open(pathname = "/dev/fd0", O_RDWR, 0666) failed, Read-only file
system (30, EROFS) because the floppy disk has the write protect
tab set
```

...because a CD-ROM is not writable
...because the memory card has the write protect tab set
...because the ½ inch magnetic tape does not have a write ring

**rename**

The *rename*(2) system call is used to change the location or name of a file, moving it between directories if required. If the destination pathname already exists it will be atomically replaced, so that there is no point at which another process attempting to access it will find it missing.

There are limitations, however: you can only rename a directory on top of another directory if the destination directory is not empty.

```
rename(oldpath = "foo", newpath = "bar") failed, Directory not
empty (39, ENOTEMPTY) because newpath is not an empty directory;
that is, it contains entries other than "." and ".."
```

You can't rename a directory on top of a non-directory, either.

```
rename(oldpath = "foo", newpath = "bar") failed, Not a directory
(20, ENOTDIR) because oldpath is a directory, but newpath is a
regular file, not a directory
```

Nor is the reverse allowed

```
rename(oldpath = "foo", newpath = "bar") failed, Is a directory
(21, EISDIR) because newpath is a directory, but oldpath is a
regular file, not a directory
```

This, of course, makes the libexplain library's job more complicated, because the *unlink*(2) or *rmdir*(2) system call is called implicitly by *rename*(2), and so all of the *unlink*(2) or *rmdir*(2) errors must be detected and handled, as well.

**dup2**

The *dup2*(2) system call is used to create a second file descriptor that references the same object as the first file descriptor. Typically this is used to implement shell input and output redirection.

The fun thing is that, just as *rename*(2) can atomically rename a file on top of an existing file and remove the old file, *dup2*(2) can do this onto an already-open file descriptor.

Once again, this makes the libexplain library's job more complicated, because the *close*(2) system call is called implicitly by *dup2*(2), and so all of *close*(2)'s errors must be detected and handled, as well.

## ADVENTURES IN IOCTL SUPPORT

The *ioctl*(2) system call provides device driver authors with a way to communicate with user-space that doesn't fit within the existing kernel API. See *ioctl_list*(2).

### Decoding Request Numbers

From a cursory look at the *ioctl*(2) interface, there would appear to be a large but finite number of possible *ioctl*(2) requests. Each different *ioctl*(2) request is effectively another system call, but without any type-safety at all − the compiler can't help a programmer get these right. This was probably the motivation behind *tcflush*(3) and friends.

The initial impression is that you could decode *ioctl*(2) requests using a huge switch statement. This turns out to be infeasible because one very rapidly discovers that it is impossible to include all of the necessary system headers defining the various *ioctl*(2) requests, because they have a hard time playing nicely with each other.

A deeper look reveals that there is a range of "private" request numbers, and device driver authors are encouraged to use them. This means that there is a far larger possible set of requests, with ambiguous request numbers, than are immediately apparent. Also, there are some historical ambiguities as well.

We already knew that the switch was impractical, but now we know that to select the appropriate request name and explanation we must consider not only the request number but also the file descriptor.

The implementation of *ioctl*(2) support within the libexplain library is to have a table of pointers to *ioctl*(2) request descriptors. Each of these descriptors includes an optional pointer to a disambiguation function.

Each request is actually implemented in a separate source file, so that the necessary include files are relieved of the obligation to play nicely with others.

### Representation

The philosophy behind the libexplain library is to provide as much information as possible, including an accurate representation of the system call. In the case of *ioctl*(2) this means printing the correct request number (by name) and also a correct (or at least useful) representation of the third argument.

The *ioctl*(2) prototype looks like this:

```
int ioctl(int fildes, int request, ...);
```

which should have your type-safety alarms going off. Internal to [e]glibc, this is turned into a variety of forms:

```
int __ioctl(int fildes, int request, long arg);
int __ioctl(int fildes, int request, void *arg);
```

and the Linux kernel syscall interface expects

```
asmlinkage long sys_ioctl(unsigned int fildes, unsigned int
request, unsigned long arg);
```

The extreme variability of the third argument is a challenge, when the libexplain library tries to print a representation of that third argument. However, once the request number has been disambiguated, each entry in the the libexplain library's ioctl table has a custom `print_data` function (OO done manually).

**Explanations**

There are fewer problems determining the explanation to be used. Once the request number has been disambiguated, each entry in the libexplain library's ioctl table has a custom `print_explanation` function (again, OO done manually).

Unlike section 2 and section 3 system calls, most *ioctl*(2) requests have no errors documented. This means, to give good error descriptions, it is necessary to read kernel sources to discover

- what *errno*(3) values may be returned, and
- the cause of each error.

Because of the OO nature of function call dispatching withing the kernel, you need to read *all* sources implementing that *ioctl*(2) request, not just the generic implementation. It is to be expected that different kernels will have different error numbers and subtly different error causes.

**EINVAL** *vs* **ENOTTY**

The situation is even worse for *ioctl*(2) requests than for system calls, with EINVAL and ENOTTY both being used to indicate that an *ioctl*(2) request is inappropriate in that context, and occasionally ENOSYS, ENOTSUP and EOPNOTSUPP (meant to be used for sockets) as well. There are comments in the Linux kernel sources that seem to indicate a progressive cleanup is in progress. For extra chaos, BSD adds ENOIOCTL to the confusion.

As a result, attention must be paid to these error cases to get them right, particularly as the EINVAL could also be referring to problems with one or more system call arguments.

**intptr_t**

The C99 standard defines an integer type that is guaranteed to be able to hold any pointer without representation loss.

The above function syscall prototype would be better written

```
long sys_ioctl(unsigned int fildes, unsigned int request, intptr_t
arg);
```

The problem is the cognitive dissonance induced by device-specific or file-system-specific *ioctl*(2) implementations, such as:

```
long vfs_ioctl(struct file *filp, unsigned int cmd, unsigned long
arg);
```

The majority of *ioctl*(2) requests actually have an `int *arg` third argument. But having it declared `long` leads to code treating this as `long *arg`. This is harmless on 32-bits (`sizeof(long) == sizeof(int)`) but nasty on 64-bits (`sizeof(long) != sizeof(int)`). Depending on the endian-ness, you do or don't get the value you expect, but you *always* get a memory scribble or stack scribble as well.

Writing all of these as

```
int ioctl(int fildes, int request, ...);
int __ioctl(int fildes, int request, intptr_t arg);
long sys_ioctl(unsigned int fildes, unsigned int request, intptr_t
arg);
long vfs_ioctl(struct file *filp, unsigned int cmd, intptr_t arg);
```

emphasizes that the integer is only an integer to represent a quantity that is almost always an unrelated pointer type.

## CONCLUSION

Use libexplain, your users will like it.

## COPYRIGHT

## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

GPL - GNU General Public License

## DESCRIPTION

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007



Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program -- to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product,

and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing

the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to

infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF

SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and a brief idea of what it does.*
> Copyright (C) *year name of author*
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

> <program> Copyright (C) <year> <name of author>
> This program comes with ABSOLUTELY NO WARRANTY; for details type "show w". This is free software, and you are welcome to redistribute it under certain conditions; type "show c" for details.

The hypothetical commands "show w" and "show c" should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

## NAME

libexplain − Explain errno values returned by libc functions

## SYNOPSIS

cc ... −lexplain;

#include <libexplain/libexplain.h>

## DESCRIPTION

The libexplain library exists to give explanations of error reported by system calls. The error message returned by *strerror*(3) tend to be quite cryptic. By providing a specific error report for each system call, a more detailed error message is possible, usually identifying and describing the specific cause from amongst the numerous meanings each *errno* value maps to.

### Race Condition

The explanation of the cause of an error is dependent on the environment of the error to remain unchanged, so that when libexplain gets around to looking for the cause, the cause is still there. On a running system, and particularly a multi-user system, this is not always possible.

If an incorrect explanation is provided, it is possible that the cause is no longer present.

### Compiling

Assuming the library header files has been installed into `/usr/include`, and the library files have been installed into `/usr/lib`, compiling against libexplain requires no special `−I` options.

When linking your pograms, add `−lexplain` to the list of libraries at the end of your link line.

```
cc ... −lexplain
```

When you configure your package with GNU Autoconf, you need the large file support macro

AC_SYS_LARGEFILE

If you aren't using GNU Autoconf, you will have to work out the needed large file support requirements yourdelf.

There is a *pkg-config*(1) package for you to use, too:

CFLAGS="$CFLAGS ‘pkg−config libexplain −−cflags‘" LIBS="$LIBS ‘pkg−config libexplain −−libs‘"

This can make figuring out the command line requirements much easier.

### Environment Variable

The `EXPLAIN_OPTIONS` environment variable may be used to control some of the content in the messages. Options are separated by comma (“,”) characters.

There are three ways to set an option:

1. The form “*name=value*” may be used explicitly. The values “true” and “false” are used for boolean options.
2. An option name alone is interpreted to mean “*name*=true”.
3. The form “no-*name*” is interpreted to mean “*name*=false”.

The following options are available:

debug
: Additional debugging messages for libexplain developers. Not generally useful to clients of the library.
Default: false.

extra-device-info
: Additional information for block and character special devices is printed when naming a file and its type.
Default: true

numeric-errno
This option includes the numeric *errno* value in the message, *e.g.* “(2, ENOENT)” rather than “(ENOENT)”. Disabling this option is generally of use in automated testing, to prevent UNIX dialect differences from producing false negatives.
Default: true

dialect-specific
This controls the presence of explanatory text specific to a particular UNIX dialect. It also suppresses printing system specific maximums. Disabling this option is generally of use in automated testing, to prevent UNIX dialect differences from producing false negatives.
Default: true.

hanging-indent
This controls the hanging indent depth used for error message wrapping. By default no hanging indent is used, but this can sometimes obfuscate the end of one error message and the beginning of another. A hanging indent results in continuation lines starting with white spoace, similar to RFC822 headers. A value of 0 means no hanging indent (all lines flush with left margin). A common value to use is 4: it doesn’t consume to much of each line, and it is a clear indent. The program may choose to override the environment variable using the *explain_option_hanging_indent_set*(3) function. The hanging indent is limited to 10% of the terminal width.
Default: 0

internal-strerror
This option controls the source of system eror message texts. If false, it uses *strerorP(3) for the text. If true, it uses internal string for the text. This is mostly of use for automated testing, to avoid false negatives induced by inconsistencies across Unix implementations.*
*Default: false.*

*program-name*
This option controls the inclusion of the program name at the start of error messages, by the explain_*_or_die and explain_*_on_error functions. This helps users understand which command is throwing the error. Disabling this option may be of some interest to script writers. Program developers can use the *explain_program_name_set*(3) function to set the name of the command, if they wish to override the name that libexplain would otherwise obtain from the operating system. Program developers can use the *explain_program_name_assemble*(3) function to trump this option.
Default: true.

symbolic-mode-bits
This option controls how permission mode bits are represented in error messages. Setting this option to true will cause symbolic names to be printed (*e.g.* S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH). Setting this option to false will cause octal values to be printed (*e.g.* 0644).
Default: false.

## Supported System Calls

Each supported system call has its own *man* page.

*explain_accept*(3)
Explain *accept*(2) errors

*explain_accept_or_die*(3)
accept a connection on a socket and report errors

*explain_accept4*(3)
Explain *accept4*(2) errors

*explain_accept4_or_die*(3)
accept a connection on a socket and report errors

*explain_access*(3)
Explain *access*(2) errors

*explain_access_or_die*(3)
check permissions for a file and report errors

*explain_acct*(3)
Explain *acct*(2) errors

*explain_acct_or_die*(3)
process accounting control and report errors

*explain_adjtime*(3)
Explain *adjtime*(2) errors

*explain_adjtime_or_die*(3)
smoothly tune kernel clock and report errors

*explain_adjtimex*(3)
Explain *adjtimex*(2) errors

*explain_adjtimex_or_die*(3)
tune kernel clock and report errors

*explain_bind*(3)
Explain *bind*(2) errors

*explain_bind_or_die*(3)
bind a name to a socket and report errors

*explain_calloc*(3)
Explain *calloc*(3) errors

*explain_calloc_or_die*(3)
Allocate and clear memory and report errors

*explain_chdir*(3)
Explain *chdir*(2) errors

*explain_chdir_or_die*(3)
change working directory and report errors

*explain_chmod*(3)
Explain *chmod*(2) errors

*explain_chmod_or_die*(3)
change permissions of a file and report errors

*explain_chown*(3)
Explain *chown* errors

*explain_chown_or_die*(3)
change ownership of a file and report errors

*explain_chroot*(3)
Explain *chroot*(2) errors

*explain_chroot_or_die*(3)
change root directory and report errors

*explain_close*(3)
Explain *close*(2) errors

*explain_close_or_die*(3)
close a file descriptor and report errors

*explain_closedir*(3)
Explain *closedir*(3) errors

*explain_closedir_or_die*(3)
close a directory and report errors

*explain_connect*(3)
Explain *connect*(2) errors

*explain_connect_or_die*(3)
initiate a connection on a socket and report errors

*explain_creat*(3)
Explain *creat*(2) errors

*explain_creat_or_die*(3)
create and open a file and report errors

*explain_dirfd*(3)
Explain *dirfd*(3) errors

*explain_dirfd_or_die*(3)
get directory stream file descriptor and report errors

*explain_dup*(3)
Explain *dup*(2) errors

*explain_dup_or_die*(3)
duplicate a file descriptor and report errors

*explain_dup2*(3)
Explain *dup2*(2) errors

*explain_dup2_or_die*(3)
duplicate a file descriptor and report errors

*explain_eventfd*(3)
Explain *eventfd*(2) errors

*explain_eventfd_or_die*(3)
create a file descriptor for event notification and report errors

*explain_execlp*(3)
Explain *execlp*(3) errors

*explain_execlp_or_die*(3)
execute a file and report errors

*explain_execv*(3)
Explain *execv*(3) errors

*explain_execv_or_die*(3)
execute a file and report errors

*explain_execve*(3)
Explain *execve*(2) errors

*explain_execve_or_die*(3)
execute program and report errors

*explain_execvp*(3)
Explain *execvp*(3) errors

*explain_execvp_or_die*(3)
execute program and report errors

*explain_exit*(3)
print an explanation of exit status before exiting

*explain_fchdir*(3)
Explain *fchdir*(2) errors

*explain_fchmod*(3)
Explain *fchmod*(2) errors

*explain_fchmod_or_die*(3)
change permissions of a file and report errors

*explain_fchown*(3)
Explain *fchown*(2) errors

*explain_fchown_or_die*(3)
change ownership of a file and report errors

*explain_fclose*(3)
Explain *fclose*(2) errors

*explain_fclose_or_die*(3)
close a stream and report errors

*explain_fcntl*(3)
Explain *fcntl*(2) errors

*explain_fcntl_or_die*(3)
Manipulate a file descriptor and report errors

*explain_fdopen*(3)
Explain *fdopen*(3) errors

*explain_fdopen_or_die*(3)
stream open function and report errors

*explain_fdopendir*(3)
Explain *fdopendir*(3) errors

*explain_fdopendir_or_die*(3)
open a directory and report errors

*explain_feof*(3)
Explain *feof*(3) errors

*explain_feof_or_die*(3)
check and reset stream status and report errors

*explain_ferror*(3)
Explain *ferror*(3) errors

*explain_ferror_or_die*(3)
check stream status and report errors

*explain_fflush*(3)
Explain *fflush*(3) errors

*explain_fflush_or_die*(3)
flush a stream and report errors

*explain_fgetc*(3)
Explain *fgetc*(3) errors

*explain_fgetc_or_die*(3)
input of characters and report errors

*explain_fgetpos*(3)
Explain *fgetpos*(3) errors

*explain_fgetpos_or_die*(3)
reposition a stream and report errors

*explain_fgets*(3)
Explain *fgets*(3) errors

*explain_fgets_or_die*(3)
input of strings and report errors

*explain_fileno*(3)
Explain *fileno*(3) errors

*explain_fileno_or_die*(3)
check and reset stream status and report errors

*explain_flock*(3)
Explain *flock*(2) errors

*explain_flock_or_die*(3)
apply or remove an advisory lock on an open file and report errors

*explain_fopen*(3)
Explain *fopen*(3) errors

*explain_fopen_or_die*(2)
open files and report errors

*explain_fork*(3)
Explain *fork*(2) errors

*explain_fork_or_die*(3)
create a child process and report errors

*explain_fpathconf*(3)
Explain *fpathconf*(3) errors

*explain_fpathconf_or_die*(3)
get configuration values for files and report errors

*explain_fprintf*(3)
Explain *fprintf*(3) errors

*explain_fprintf_or_die*(3)
formatted output conversion and report errors

*explain_fpurge*(3)
Explain *fpurge*(3) errors

*explain_fpurge_or_die*(3)
purge a stream and report errors

*explain_fputc*(3)
Explain *fputc*(3) errors

*explain_fputc_or_die*(3)
output of characters and report errors

*explain_fputs*(3)
Explain *fputs*(3) errors

*explain_fputs_or_die*(3)
write a string to a stream and report errors

*explain_fread*(3)
Explain *fread*(3) errors

*explain_fread_or_die*(3)
binary stream input and report errors

*explain_freopen*(3)
Explain *freopen*(3) errors

*explain_freopen_or_die*(3)
open files and report errors

*explain_fseek*(3)
Explain *fseek*(3) errors

*explain_fseek_or_die*(3)
reposition a stream and report errors

*explain_fsetpos*(3)
Explain *fsetpos*(3) errors

*explain_fsetpos_or_die*(3)
reposition a stream and report errors

*explain_fstat*(3)
Explain *fstat*(3) errors

*explain_fstat_or_die*(3)
get file status and report errors

*explain_fstatfs*(3)
Explain *fstatfs*(2) errors

*explain_fstatfs_or_die*(3)
get file system statistics and report errors

*explain_fstatvfs*(3)
Explain *fstatvfs*(2) errors

*explain_fstatvfs_or_die*(3)
get file system statistics and report errors

*explain_fsync*(3)
Explain *fsync*(2) errors

*explain_fsync_or_die*(3)
synchronize a file's in-core state with storage device and report errors

*explain_ftell*(3)
Explain *ftell*(3) errors

*explain_ftell_or_die*(3)
get stream position and report errors

*explain_ftime*(3)
Explain *ftime*(3) errors

*explain_ftime_or_die*(3)
return date and time and report errors

*explain_ftruncate*(3)
Explain *ftruncate*(2) errors

*explain_ftruncate_or_die*(3)
truncate a file to a specified length and report errors

*explain_futimes*(3)
Explain *futimes*(3) errors

*explain_futimes_or_die*(3)
Execute *futimes*(3) and report errors

*explain_fwrite*(3)
Explain *fwrite*(3) errors

*explain_fwrite_or_die*(3)
binary stream output and report errors

*explain_getaddrinfo*(3)
Explain *getaddrinfo*(3) errors

*explain_getaddrinfo_or_die*(3)
network address and and report errors

*explain_getc*(3)
Explain *getc*(3) errors

*explain_getc_or_die*(3)
input of characters and report errors

*explain_getchar*(3)
Explain *getchar*(3) errors

*explain_getchar_or_die*(3)
input of characters and report errors

*explain_getcwd*(3)
Explain *getcwd*(2) errors

*explain_getdomainname*(3)
Explain *getdomainname*(2) errors

*explain_getdomainname_or_die*(3)
get domain name and report errors

*explain_getgroups*(3)
Explain *getgroups*(2) errors

*explain_getgroups_or_die*(3)
get list of supplementary group IDs and report errors

*explain_getcwd_or_die*(3)
Get current working directory and report errors

*explain_gethostbyname*(3)
Explain *gethostbyname*(3) errors

*explain_gethostbyname_or_die*(3)
get host address given host name and report errors

*explain_gethostname*(3)
Explain *gethostname*(2) errors

*explain_gethostname_or_die*(3)
get hostname and report errors

*explain_getpeername*(3)
Explain *getpeername*(2) errors

*explain_getpeername_or_die*(3)
Execute*getpeername*(2) and report errors

*explain_getpgid*(3)
Explain *getpgid*(2) errors

*explain_getpgid_or_die*(3)
get process group and report errors

*explain_getpgrp*(3)
Explain *getpgrp*(2) errors

*explain_getpgrp_or_die*(3)
get process group and report errors

*explain_getresgid*(3)
Explain *getresgid*(2) errors

*explain_getresgid_or_die*(3)
get real, effective and saved group IDs and report errors

*explain_getresuid*(3)
Explain *getresuid*(2) errors

*explain_getresuid_or_die*(3)
get real, effective and saved user IDs and report errors

*explain_getrlimit*(3)
Explain *getrlimit*(2) errors

*explain_getrlimit_or_die*(3)
get resource limits and report errors

*explain_getrusage*(3)
Explain *getrusage*(2) errors

*explain_getrusage_or_die*(3)
get resource usage and report errors

*explain_getsockname*(3)
Explain *getsockname*(2) errors

*explain_getsockname_or_die*(3)
Execute *getsockname*(2) and report errors

*explain_getsockopt*(3)
Explain *getsockopt*(2) errors

*explain_getsockopt_or_die*(3)
Execute *getsockopt*(2) and report errors

*explain_gettimeofday*(3)
Explain *gettimeofday*(2) errors

*explain_gettimeofday_or_die*(3)
get time and report errors

*explain_getw*(3)
Explain *getw*(3) errors

*explain_getw_or_die*(3)
input a word (int) and report errors

*explain_ioctl*(3)
Explain *ioctl*(2) errors

*explain_ioctl_or_die*(3)
Execute *ioctl*(2) and report errors

*explain_kill*(3)
Explain *kill*(2) errors

*explain_kill_or_die*(3)
send signal to a process and report errors

*explain_lchmod*(3)
Explain *lchmod*(2) errors

*explain_lchmod_or_die*(3)
change permissions of a file and report errors

*explain_lchown*(3)
Explain *lchown*(2) errors

*explain_lchown_or_die*(3)
change ownership of a file and report errors

*explain_link*(3)
Explain *link*(2) errors

*explain_link_or_die*(3)
make a new name for a file and report errors

*explain_listen*(3)
Explain *listen*(2) errors

*explain_listen_or_die*(3)
listen for connections on a socket and report errors

*explain_lseek*(3)
Explain *lseek*(2) errors

*explain_lseek_or_die*(3)
reposition file offset and report errors

*explain_lstat*(3)
Explain *lstat*(2) errors

*explain_lstat_or_die*(3)
get file status and report errors

*explain_malloc*(3)
Explain *malloc*(3) errors

*explain_malloc_or_die*(3)
Execute *malloc*(3) and report errors

*explain_mkdir*(3)
Explain *mkdir*(2) errors

*explain_mkdir_or_die*(3)
create directory and report errors

*explain_mkdtemp*(3)
Explain *mkdtemp*(3) errors

*explain_mkdtemp_or_die*(3)
create a unique temporary directory and report errors

*explain_mknod*(3)
Explain *mknod*(2) errors

*explain_mknod_or_die*(3)
create a special or ordinary file and report errors

*explain_mkostemp*(3)
Explain *mkostemp*(3) errors

*explain_mkostemp_or_die*(3)
create a unique temporary file and report errors

*explain_mkstemp*(3)
Explain *mkstemp*(3) errors

*explain_mkstemp_or_die*(3)
create a unique temporary file and report errors

*explain_mktemp*(3)
Explain *mktemp*(3) errors

*explain_mktemp_or_die*(3)
make a unique temporary filename and report errors

*explain_mmap*(3)
Explain *mmap*(2) errors

*explain_mmap_or_die*(3)
map file or device into memory and report errors

*explain_munmap*(3)
Explain *munmap*(2) errors

*explain_munmap_or_die*(3)
unmap a file or device from memory and report errors

*explain_nice*(3)
Explain *nice*(2) errors

*explain_nice_or_die*(3)
change process priority and report errors

*explain_open*(3)
Explain *open*(2) errors

*explain_open_or_die*(3)
open files and report errors

*explain_opendir*(3)
Explain *opendir*(3) errors

*explain_opendir_or_die*(3)
open a directory and report errors

*explain_pathconf*(3)
Explain *pathconf*(3) errors

*explain_pathconf_or_die*(3)
get configuration values for files and report errors

*explain_pclose*(3)
Explain *pclose*(3) errors

*explain_pclose_or_die*(3)
Execute *pclose*(3) and report errors

*explain_pipe*(3)
Explain *pipe*(2) errors

*explain_pipe_or_die*(3)
Execute *pipe*(2) and report errors

*explain_poll*(3)
Explain *poll*(2) errors

*explain_poll_or_die*(3)
wait for some event on a file descriptor and report errors

*explain_popen*(3)
Explain *popen*(3) errors

*explain_popen_or_die*(3)
Execute *popen*(3) and report errors

*explain_pread*(3)
Explain *pread*(2) errors

*explain_pread_or_die*(3)
read from a file descriptor at a given offset and report errors

*explain_printf*(3)
Explain *printf*(3) errors

*explain_printf_or_die*(3)
formatted output conversion and report errors

*explain_ptrace*(3)
Explain *ptrace*(2) errors

*explain_ptrace_or_die*(3)
process trace and report errors

*explain_putc*(3)
Explain *putc*(3) errors

*explain_putc_or_die*(3)
output of characters and report errors

*explain_putchar*(3)
Explain *putchar*(3) errors

*explain_putchar_or_die*(3)
output of characters and report errors

*explain_putenv*(3)
Explain *putenv*(3) errors

*explain_putenv_or_die*(3)
change or add an environment variable and report errors

*explain_puts*(3)
Explain *puts*(3) errors

*explain_puts_or_die*(3)
write a string and a trailing newline to stdout and report errors

*explain_putw*(3)
Explain *putw*(3) errors

*explain_putw_or_die*(3)
output a word (int) and report errors

*explain_pwrite*(3)
Explain *pwrite*(2) errors

*explain_pwrite_or_die*(3)
write to a file descriptor at a given offset and report errors

*explain_raise*(3)
Explain *raise*(3) errors

*explain_raise_or_die*(3)
send a signal to the caller and report errors

*explain_read*(3)
Explain *read*(2) errors

*explain_read_or_die*(3)
read from a file descriptor and report errors

*explain_readdir*(3)
Explain *readdir*(3) errors

*explain_readdir_or_die*(3)
read a directory and report errors

*explain_readlink*(3)
Explain *readlink*(2) errors

*explain_readlink_or_die*(3)
read value of a symbolic link and report errors

*explain_readv*(3)
Explain *readv*(2) errors

*explain_readv_or_die*(3)
read data into multiple buffers and report errors

*explain_realloc*(3)
Explain *realloc*(3) errors

*explain_realloc_or_die*(3)
Execute *realloc*(3) and report errors

*explain_realpath*(3)
Explain *realpath*(3) errors

*explain_realpath_or_die*(3)
return the canonicalized absolute pathname and report errors

*explain_rename*(3)
Explain *rename*(2) errors

*explain_rename_or_die*(3)
change the name or location of a file and report errors

*explain_rmdir*(3)
Explain *rmdir*(2) errors

*explain_rmdir_or_die*(3)
delete a directory and report errors

*explain_select*(3)
Explain *select*(2) errors

*explain_select_or_die*(3)
execute *select*(2) and report errors

*explain_setbuf*(3)
Explain *setbuf*(3) errors

*explain_setbuffer*(3)
Explain *setbuffer*(3) errors

*explain_setbuffer_or_die*(3)
stream buffering operations and report errors

*explain_setbuf_or_die*(3)
set stream buffer and report errors

*explain_setdomainname*(3)
Explain *setdomainname*(2) errors

*explain_setdomainname_or_die*(3)
set domain name and report errors

*explain_setenv*(3)
Explain *setenv*(3) errors

*explain_setenv_or_die*(3)
change or add an environment variable and report errors

*explain_setgid*(3)
Explain *setgid*(2) errors

*explain_setgid_or_die*(3)
set group identity and report errors

*explain_setgroups*(3)
Explain *setgroups*(2) errors

*explain_setgroups_or_die*(3)
get list of supplementary group IDs and report errors

*explain_sethostname*(3)
Explain *sethostname*(2) errors

*explain_sethostname_or_die*(3)
set hostname and report errors

*explain_setlinebuf*(3)
Explain *setlinebuf*(3) errors

*explain_setlinebuf_or_die*(3)
stream buffering operations and report errors

*explain_setpgid*(3)
Explain *setpgid*(2) errors

*explain_setpgid_or_die*(3)
set process group and report errors

*explain_setpgrp*(3)
Explain *setpgrp*(2) errors

*explain_setpgrp_or_die*(3)
set process group and report errors

*explain_setregid*(3)
Explain *setregid*(2) errors

*explain_setregid_or_die*(3)
set real and/or effective group ID and report errors

*explain_setreuid*(3)
Explain *setreuid*(2) errors

*explain_setreuid_or_die*(3)
set the real and effective user ID and report errors

*explain_setresgid*(3)
Explain *setresgid*(2) errors

*explain_setresgid_or_die*(3)
set real, effective and saved group ID and report errors

*explain_setresuid*(3)
Explain *setresuid*(2) errors

*explain_setresuid_or_die*(3)
set real, effective and saved user ID and report errors

*explain_setreuid*(3)
Explain *setreuid*(2) errors

*explain_setreuid_or_die*(3)
set real and/or effective user ID and report errors

*explain_setsid*(3)
Explain *setsid*(2) errors

*explain_setsid_or_die*(3)
creates a session and sets the process group ID and report errors

*explain_setsockopt*(3)
Explain *setsockopt*(2) errors

*explain_setsockopt_or_die*(3)
execute *setsockopt*(2) and report errors

*explain_setuid*(3)
Explain *setuid*(2) errors

*explain_setuid_or_die*(3)
set user identity and report errors

*explain_setvbuf*(3)
Explain *setvbuf*(3) errors

*explain_setvbuf_or_die*(3)
stream buffering operations and report errors

*explain_shmat*(3)
Explain *shmat*(2) errors

*explain_shmat_or_die*(3)
shared memory attach and report errors

*explain_shmctl*(3)
Explain *shmctl*(2) errors

*explain_shmctl_or_die*(3)
shared memory control and report errors

*explain_signalfd*(3)
Explain *signalfd*(2) errors

*explain_signalfd_or_die*(3)
create a file descriptor for accepting signals and report errors

*explain_socket*(3)
Explain *socket*(2) errors

*explain_socket_or_die*(3)
create an endpoint for communication and report errors

*explain_socketpair*(3)
Explain *socketpair*(2) errors

*explain_socketpair_or_die*(3)
create a pair of connected sockets and report errors

*explain_sprintf*(3)
Explain *sprintf*(3) errors

*explain_sprintf_or_die*(3)
formatted output conversion and report errors

*explain_stat*(3)
Explain *stat*(2) errors

*explain_statfs*(3)
Explain *statfs*(2) errors

*explain_statfs_or_die*(3)
get file system statistics and report errors

*explain_statvfs*(3)
Explain *statvfs*(2) errors

*explain_statvfs_or_die*(3)
get file system statistics and report errors

*explain_stime*(3)
Explain *stime*(2) errors

*explain_stime_or_die*(3)
set system time and report errors

*explain_strdup*(3)
Explain *strdup*(3) errors

*explain_strdup_or_die*(3)
duplicate a string and report errors

*explain_strndup*(3)
Explain *strndup*(3) errors

*explain_strndup_or_die*(3)
duplicate a string and report errors

*explain_strtod*(3)
Explain *strtod*(3) errors

*explain_strtod_or_die*(3)
convert string to floating-point number and report errors

*explain_strtof*(3)
Explain *strtof*(3) errors

*explain_strtof_or_die*(3)
convert string to floating-point number and report errors

*explain_strtol*(3)
Explain *strtol*(3) errors

*explain_strtol_or_die*(3)
convert a string to a long integer and report errors

*explain_strtold*(3)
Explain *strtold*(3) errors

*explain_strtold_or_die*(3)
convert string to floating-point number and report errors

*explain_strtoll*(3)
Explain *strtoll*(3) errors

*explain_strtoll_or_die*(3)
convert a string to a long long integer and report errors

*explain_strtoul*(3)
Explain *strtoul*(3) errors

*explain_strtoul_or_die*(3)
convert a string to a long long integer and report errors

*explain_strtoull*(3)
Explain *strtoull*(3) errors

*explain_strtoull_or_die*(3)
convert a string to an unsigned long long integer and report errors

*explain_symlink*(3)
Explain *symlink*(2) errors

*explain_symlink_or_die*(3)
make a new name for a file and report errors

*explain_system*(3)
Explain *system*(3) errors

*explain_system_or_die*(3)
execute a shell command and report errors

*explain_tcdrain*(3)
Explain *tcdrain*(3) errors

*explain_tcdrain_or_die*(3)
Execute *tcdrain*(3) and report errors

*explain_tcflow*(3)
Explain *tcflow*(3) errors

*explain_tcflow_or_die*(3)
Execute *tcflow*(3) and report errors

*explain_tcflush*(3)
Explain *tcflush*(3) errors

*explain_tcflush_or_die*(3)
discard terminal data and report errors

*explain_tcgetattr*(3)
Explain *tcgetattr*(3) errors

*explain_tcgetattr_or_die*(3)
get terminal parameters and report errors

*explain_tcsendbreak*(3)
Explain *tcsendbreak*(3) errors

*explain_tcsendbreak_or_die*(3)
send terminal line break and report errors

*explain_tcsetattr*(3)
Explain *tcsetattr*(3) errors

*explain_tcsetattr_or_die*(3)
set terminal attributes and report errors

*explain_telldir*(3)
Explain *telldir*(3) errors

*explain_telldir_or_die*(3)
return current location in directory stream and report errors

*explain_tempnam*(3)
Explain *tempnam*(3) errors

*explain_tempnam_or_die*(3)
create a name for a temporary file and report errors

*explain_time*(3)
Explain *time*(2) errors

*explain_time_or_die*(3)
get time in seconds and report errors

*explain_timerfd_create*(3)
Explain *timerfd_create*(2) errors

*explain_timerfd_create_or_die*(3)
timers that notify via file descriptors and report errors

*explain_tmpfile*(3)
Explain *tmpfile*(3) errors

*explain_tmpfile_or_die*(3)
create a temporary file and report errors

*explain_tmpnam*(3)
Explain *tmpnam*(3) errors

*explain_tmpnam_or_die*(3)
create a name for a temporary file and report errors

*explain_truncate*(3)
Explain *truncate*(2) errors

*explain_truncate_or_die*(3)
truncate a file to a specified length and report errors

*explain_ungetc*(3)
Explain *ungetc*(3) errors

*explain_ungetc_or_die*(3)
push a character back to a stream and report errors

*explain_unlink*(3)
Explain *unlink*(2) errors

*explain_unlink_or_die*(3)
delete a file and report errors

*explain_unsetenv*(3)
Explain *unsetenv*(3) errors

*explain_unsetenv_or_die*(3)
remove an environment variable and report errors

*explain_ustat*(3)
Explain *ustat*(2) errors

*explain_ustat_or_die*(3)
get file system statistics and report errors

*explain_utime*(3)
Explain *utime*(2) errors

*explain_utime_or_die*(3)
change file last access and modification times and report errors

*explain_utimens*(3)
Explain *utimens*(2) errors

*explain_utimens_or_die*(3)
change file last access and modification times and report errors

*explain_utimensat*(3)
Explain *utimensat*(2) errors

*explain_utimensat_or_die*(3)
change file timestamps with nanosecond precision and report errors

*explain_utimes*(3)
Explain *utimes*(2) errors

*explain_utimes_or_die*(3)
change file last access and modification times and report errors

*explain_vfork*(3)
Explain *vfork*(2) errors

*explain_vfork_or_die*(3)
create a child process and block parent and report errors

*explain_vfprintf*(3)
Explain *vfprintf*(3) errors

*explain_vfprintf_or_die*(3)
formatted output conversion and report errors

*explain_vprintf*(3)
Explain *vprintf*(3) errors

*explain_vprintf_or_die*(3)
formatted output conversion and report errors

*explain_vsnprintf*(3)
Explain *vsnprintf*(3) errors

*explain_vsnprintf_or_die*(3)
formatted output conversion and report errors

*explain_snprintf*(3)
Explain *snprintf*(3) errors

*explain_snprintf_or_die*(3)
formatted output conversion and report errors

*explain_vsprintf*(3)
Explain *vsprintf*(3) errors

*explain_vsprintf_or_die*(3)
formatted output conversion and report errors

*explain_wait*(3)
Explain *wait*(2) errors

*explain_wait_or_die*(3)
wait for process to change state and report errors

*explain_wait3*(3)
Explain *wait3*(2) errors

*explain_wait3_or_die*(3)
wait for process to change state and report errors

*explain_wait4*(3)
Explain *wait4*(2) errors

*explain_wait4_or_die*(3)
wait for process to change state and report errors

*explain_waitpid*(3)
Explain *waitpid*(2) errors

*explain_waitpid_or_die*(3)
wait for process to change state and report errors

*explain_write*(3)
Explain *write*(2) errors

*explain_write_or_die*(3)
write to a file descriptor and report errors

*explain_writev*(3)
Explain *writev*(2) errors

*explain_writev_or_die*(3)
write data from multiple buffers and report errors

There are plans for additional coverage. This list is expected to expand in later releases of this library.

## SEE ALSO

*errno*(3) number of last error

*perror*(3)
print a system error message

*strerror*(3)
return string describing error number

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_accept − explain accept(2) errors

## SYNOPSIS

#include <libexplain/accept.h>

const char *explain_accept(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);
const char *explain_errno_accept(int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);
void explain_message_accept(char *message, int message_size, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addrlen);
void explain_message_errno_accept(char *message, int message_size, int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *accept*(2) system call.

### explain_accept

const char *explain_accept(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_accept** function is used to obtain an explanation of an error returned by the *accept*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (accept(fildes, sock_addr, sock_addr_size) < 0)
{
    fprintf(stderr, "%s\n", explain_accept(fildes, sock_addr,
        sock_addr_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_accept_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *accept*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *accept*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *accept*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_accept

const char *explain_errno_accept(int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_errno_accept** function is used to obtain an explanation of an error returned by the *accept*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (accept(fildes, sock_addr, sock_addr_size) < 0)
{
    int err = errno;
```

```
        fprintf(stderr, "%s\n", explain_errno_accept(err, fildes, sock_addr,
            sock_addr_size));
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_accept_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *accept*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *accept*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *accept*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_accept**

void explain_message_accept(char *message, int message_size, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_message_accept** function may be used to obtain an explanation of an error returned by the *accept*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
    if (accept(fildes, sock_addr, sock_addr_size) < 0)
    {
        char message[3000];
        explain_message_accept(message, sizeof(message), fildes, sock_addr,
            sock_addr_size);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_accept_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *accept*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *accept*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *accept*(2) system call.

**explain_message_errno_accept**

void explain_message_errno_accept(char *message, int message_size, int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_message_errno_accept** function may be used to obtain an explanation of an error returned by the *accept*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (accept(fildes, sock_addr, sock_addr_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_accept(message, sizeof(message), err, fildes,
        sock_addr, sock_addr_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_accept_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *accept*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *accept*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *accept*(2) system call.

## SEE ALSO

*accept*(2)
accept a connection on a socket

*explain_accept_or_die*(3)
accept a connection on a socket and report errors

## COPYRIGHT

## NAME

explain_accept4 − explain accept4(2) errors

## SYNOPSIS

#include <libexplain/accept4.h>

const char *explain_accept4(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);
const char *explain_errno_accept4(int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);
void explain_message_accept4(char *message, int message_size, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);
void explain_message_errno_accept4(char *message, int message_size, int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *accept4*(2) system call.

### explain_accept4

const char *explain_accept4(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);

The **explain_accept4** function is used to obtain an explanation of an error returned by the *accept4*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *accept4*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *accept4*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *accept4*(2) system call.

*flags* The original flags, exactly as passed to the *accept4*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = accept4(fildes, sock_addr, sock_addr_size, flags);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_accept4(fildes, sock_addr,
   sock_addr_size, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_accept4_or_die*(3) function.

### explain_errno_accept4

const char *explain_errno_accept4(int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);

The **explain_errno_accept4** function is used to obtain an explanation of an error returned by the *accept4*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *accept4*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *accept4*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *accept4*(2) system call.

*flags* The original flags, exactly as passed to the *accept4*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = accept4(fildes, sock_addr, sock_addr_size, flags);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_accept4(err, fildes,
   sock_addr, sock_addr_size, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_accept4_or_die*(3) function.

**explain_message_accept4**

void explain_message_accept4(char *message, int message_size, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);

The **explain_message_accept4** function is used to obtain an explanation of an error returned by the *accept4*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *accept4*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *accept4*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *accept4*(2) system call.

*flags* The original flags, exactly as passed to the *accept4*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = accept4(fildes, sock_addr, sock_addr_size, flags);
if (result < 0)
{
    char message[3000];
   explain_message_accept4(message, sizeof(message), fildes,
   sock_addr, sock_addr_size, flags);
    fprintf(stderr, "%s\n", message);
```

```
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_accept4_or_die*(3) function.

**explain_message_errno_accept4**

void explain_message_errno_accept4(char *message, int message_size, int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);

The **explain_message_errno_accept4** function is used to obtain an explanation of an error returned by the *accept4*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *accept4*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *accept4*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *accept4*(2) system call.

*flags* The original flags, exactly as passed to the *accept4*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = accept4(fildes, sock_addr, sock_addr_size, flags);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_accept4(message, sizeof(message), err,
   fildes, sock_addr, sock_addr_size, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_accept4_or_die*(3) function.

## SEE ALSO

*accept4*(2)
accept a connection on a socket

*explain_accept4_or_die*(3)
accept a connection on a socket and report errors

## COPYRIGHT

## NAME

explain_accept4_or_die − accept a connection on a socket and report errors

## SYNOPSIS

#include <libexplain/accept4.h>

int explain_accept4_or_die(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);
int explain_accept4_on_error(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size, int flags);

## DESCRIPTION

The **explain_accept4_or_die** function is used to call the *accept4*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_accept4*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_accept4_on_error** function is used to call the *accept4*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_accept4*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *accept4*(2) system call.

*sock_addr*
The sock_addr, exactly as to be passed to the *accept4*(2) system call.

*sock_addr_size*
The sock_addr_size, exactly as to be passed to the *accept4*(2) system call.

*flags* The flags, exactly as to be passed to the *accept4*(2) system call.

## RETURN VALUE

The **explain_accept4_or_die** function only returns on success, see *accept4*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_accept4_on_error** function always returns the value return by the wrapped *accept4*(2) system call.

## EXAMPLE

The **explain_accept4_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_accept4_or_die(fildes, sock_addr, sock_addr_size, flags);
```

## SEE ALSO

*accept4*(2)
accept a connection on a socket

*explain_accept4*(3)
explain *accept4*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_accept_or_die − accept a connection on a socket and report errors

## SYNOPSIS

#include <libexplain/accept.h>

int explain_accept_or_die(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

## DESCRIPTION

The **explain_accept_or_die** function is used to call the *accept*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_accept*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int fd = explain_accept_or_die(fildes, sock_addr, sock_addr_size);
```

*fildes* The fildes, exactly as to be passed to the *accept*(2) system call.

*sock_addr*
The sock_addr, exactly as to be passed to the *accept*(2) system call.

*sock_addr_size*
The sock_addr_size, exactly as to be passed to the *accept*(2) system call.

Returns: This function only returns on success, see *accept*(2) for more information. On failure, prints an explanation and exits.

## SEE ALSO

*accept*(2)
accept a connection on a socket

*explain_accept*(3)
explain *accept*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_access − explain access(2) errors

## SYNOPSIS

#include <libexplain/access.h>
const char *explain_access(const char *pathname, int mode);
const char *explain_errno_access(int errnum, const char *pathname, int mode);
void explain_message_access(char *message, int message_size, const char *pathname, int mode);
void explain_message_errno_access(char *message, int message_size, int errnum, const char *pathname, int mode);

## DESCRIPTION

These functions may be used to obtain explanations for *access*(2) errors.

### explain_access

const char *explain_access(const char *pathname, int mode);

The explain_access function is used to obtain an explanation of an error returned by the *access*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int fd = access(pathname, mode);
if (fd < 0)
{
    fprintf(stderr, "%s0, explain_access(pathname, mode));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *access*(2) system call.

*mode*
: The original mode, exactly as passed to the *access*(2) system call. TP 8n Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_access

const char *explain_errno_access(int errnum, const char *pathname, int mode);

The explain_errno_access function is used to obtain an explanation of an error returned by the *access*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int fd = access(pathname, mode);
if (fd < 0)
{
    int err = errno;
    fprintf(stderr, "%s0, explain_errno_access(err, pathname,
        mode));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *access*(2) system call.

*mode* The original mode, exactly as passed to the *access*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_access**

void explain_message_access(char *message, int message_size, const char *pathname, int mode);

The explain_message_access function is used to obtain an explanation of an error returned by the *access*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int fd = access(pathname, mode);
if (fd < 0)
{
    char message[3000];
    explain_message_access(message, sizeof(message), pathname,
        mode);
    fprintf(stderr, "%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *access*(2) system call.

*mode* The original mode, exactly as passed to the *access*(2) system call.

**explain_message_errno_access**

void explain_message_errno_access(char *message, int message_size, int errnum, const char *pathname, int mode);

The explain_message_errno_access function is used to obtain an explanation of an error returned by the *access*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following exameple:

```
int fd = access(pathname, mode);
if (fd < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_access(message, sizeof(message), err,
        pathname, mode);
    fprintf(stderr, "%s0, message);
    exit(EXIT_FAILURE);
```

```
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *access*(2) system call.

*mode* The original mode, exactly as passed to the *access*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_access_or_die − check permissions for a file and report errors

## SYNOPSIS

#include <libexplain/libexplain.h>
void explain_access_or_die(const char *pathname, int mode);

## DESCRIPTION

The explain_access_or_die function is used to call the *access*(2) system call and check the result. On failure it prints an explanation of the error, obtained from *explain_access*(3), and then terminates by calling `exit(EXIT_FAILURE)`.

```
explain_access_or_die(pathname, mode);
```

*pathname*
: The pathname, exactly as to be passed to the *access*(2) system call.

*mode*
: The mode, exactly as to be passed to the *access*(2) system call.

Returns: Only ever return on success. On failure process will exit.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_acct − explain acct(2) errors

## SYNOPSIS

#include <libexplain/acct.h>

const char *explain_acct(const char *pathname);
const char *explain_errno_acct(int errnum, const char *pathname);
void explain_message_acct(char *message, int message_size, const char *pathname);
void explain_message_errno_acct(char *message, int message_size, int errnum, const char *pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *acct*(2) system call.

### explain_acct

const char *explain_acct(const char *pathname);

The **explain_acct** function is used to obtain an explanation of an error returned by the *acct*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *acct*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (acct(pathname) < 0)
{
   fprintf(stderr, "%s\n", explain_acct(pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_acct_or_die*(3) function.

### explain_errno_acct

const char *explain_errno_acct(int errnum, const char *pathname);

The **explain_errno_acct** function is used to obtain an explanation of an error returned by the *acct*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *acct*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (acct(pathname) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_acct(err, pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_acct_or_die*(3) function.

**explain_message_acct**

void explain_message_acct(char *message, int message_size, const char *pathname);

The **explain_message_acct** function is used to obtain an explanation of an error returned by the *acct*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *acct*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (acct(pathname) < 0)
{
    char message[3000];
    explain_message_acct(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_acct_or_die*(3) function.

**explain_message_errno_acct**

void explain_message_errno_acct(char *message, int message_size, int errnum, const char *pathname);

The **explain_message_errno_acct** function is used to obtain an explanation of an error returned by the *acct*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *acct*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (acct(pathname) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_acct(message, sizeof(message), err,
```

```
                pathname);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_acct_or_die*(3) function.

## SEE ALSO

*acct*(2) switch process accounting on or off

*explain_acct_or_die*(3)
switch process accounting on or off and report errors

## COPYRIGHT

## NAME

explain_acct_or_die − switch process accounting on or off and report errors

## SYNOPSIS

#include <libexplain/acct.h>

void explain_acct_or_die(const char *pathname);
int explain_acct_on_error(const char *pathname))

## DESCRIPTION

The **explain_acct_or_die** function is used to call the *acct*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_acct*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_acct_on_error** function is used to call the *acct*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_acct*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *acct*(2) system call.

## RETURN VALUE

The **explain_acct_or_die** function only returns on success, see *acct*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_acct_on_error** function always returns the value return by the wrapped *acct*(2) system call.

## EXAMPLE

The **explain_acct_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_acct_or_die(pathname);
```

## SEE ALSO

*acct*(2) switch process accounting on or off

*explain_acct*(3)
: explain *acct*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_adjtime − explain adjtime(2) errors

## SYNOPSIS

#include <libexplain/adjtime.h>

const char *explain_adjtime(const struct timeval *delta, struct timeval *olddelta);
const char *explain_errno_adjtime(int errnum, const struct timeval *delta, struct timeval *olddelta);
void explain_message_adjtime(char *message, int message_size, const struct timeval *delta, struct timeval *olddelta);
void explain_message_errno_adjtime(char *message, int message_size, int errnum, const struct timeval *delta, struct timeval *olddelta);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *adjtime*(2) system call.

### explain_adjtime

const char *explain_adjtime(const struct timeval *delta, struct timeval *olddelta);

The **explain_adjtime** function is used to obtain an explanation of an error returned by the *adjtime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*delta* The original delta, exactly as passed to the *adjtime*(2) system call.

*olddelta* The original olddelta, exactly as passed to the *adjtime*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (adjtime(delta, olddelta) < 0)
{
   fprintf(stderr, "%s\n", explain_adjtime(delta, olddelta));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_adjtime_or_die*(3) function.

### explain_errno_adjtime

const char *explain_errno_adjtime(int errnum, const struct timeval *delta, struct timeval *olddelta);

The **explain_errno_adjtime** function is used to obtain an explanation of an error returned by the *adjtime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*delta* The original delta, exactly as passed to the *adjtime*(2) system call.

*olddelta* The original olddelta, exactly as passed to the *adjtime*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (adjtime(delta, olddelta) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_adjtime(err, delta,
    olddelta));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_adjtime_or_die*(3) function.

**explain_message_adjtime**

void explain_message_adjtime(char *message, int message_size, const struct timeval *delta, struct timeval *olddelta);

The **explain_message_adjtime** function is used to obtain an explanation of an error returned by the *adjtime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*delta* The original delta, exactly as passed to the *adjtime*(2) system call.

*olddelta* The original olddelta, exactly as passed to the *adjtime*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (adjtime(delta, olddelta) < 0)
{
    char message[3000];
    explain_message_adjtime(message, sizeof(message), delta,
    olddelta);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_adjtime_or_die*(3) function.

**explain_message_errno_adjtime**

void explain_message_errno_adjtime(char *message, int message_size, int errnum, const struct timeval *delta, struct timeval *olddelta);

The **explain_message_errno_adjtime** function is used to obtain an explanation of an error returned by the *adjtime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*delta* The original delta, exactly as passed to the *adjtime*(2) system call.

*olddelta* The original olddelta, exactly as passed to the *adjtime*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (adjtime(delta, olddelta) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_adjtime(message, sizeof(message), err,
   delta, olddelta);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_adjtime_or_die*(3) function.

## SEE ALSO

*adjtime*(2)
smoothly tune kernel clock

*explain_adjtime_or_die*(3)
smoothly tune kernel clock and report errors

## COPYRIGHT

## NAME

explain_adjtime_or_die − smoothly tune kernel clock and report errors

## SYNOPSIS

#include <libexplain/adjtime.h>

void explain_adjtime_or_die(const struct timeval *delta, struct timeval *olddelta);
int explain_adjtime_on_error(const struct timeval *delta, struct timeval *olddelta);

## DESCRIPTION

The **explain_adjtime_or_die** function is used to call the *adjtime*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_adjtime*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_adjtime_on_error** function is used to call the *adjtime*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_adjtime*(3) function, but still returns to the caller.

*delta* The delta, exactly as to be passed to the *adjtime*(2) system call.

*olddelta* The olddelta, exactly as to be passed to the *adjtime*(2) system call.

## RETURN VALUE

The **explain_adjtime_or_die** function only returns on success, see *adjtime*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_adjtime_on_error** function always returns the value return by the wrapped *adjtime*(2) system call.

## EXAMPLE

The **explain_adjtime_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_adjtime_or_die(delta, olddelta);
```

## SEE ALSO

*adjtime*(2)
smoothly tune kernel clock

*explain_adjtime*(3)
explain *adjtime*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_adjtimex − explain adjtimex(2) errors

## SYNOPSIS

#include <libexplain/adjtimex.h>

const char *explain_adjtimex(struct timex *data);
const char *explain_errno_adjtimex(int errnum, struct timex *data);
void explain_message_adjtimex(char *message, int message_size, struct timex *data);
void explain_message_errno_adjtimex(char *message, int message_size, int errnum, struct timex *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *adjtimex*(2) system call.

### explain_adjtimex

const char *explain_adjtimex(struct timex *data);

The **explain_adjtimex** function is used to obtain an explanation of an error returned by the *adjtimex*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *adjtimex*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = adjtimex(data);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_adjtimex(data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_adjtimex_or_die*(3) function.

### explain_errno_adjtimex

const char *explain_errno_adjtimex(int errnum, struct timex *data);

The **explain_errno_adjtimex** function is used to obtain an explanation of an error returned by the *adjtimex*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *adjtimex*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = adjtimex(data);
```

```
if (result < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_adjtimex(err, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_adjtimex_or_die*(3) function.

**explain_message_adjtimex**

void explain_message_adjtimex(char *message, int message_size, struct timex *data);

The **explain_message_adjtimex** function is used to obtain an explanation of an error returned by the *adjtimex*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *adjtimex*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = adjtimex(data);
if (result < 0)
{
    char message[3000];
    explain_message_adjtimex(message, sizeof(message), data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_adjtimex_or_die*(3) function.

**explain_message_errno_adjtimex**

void explain_message_errno_adjtimex(char *message, int message_size, int errnum, struct timex *data);

The **explain_message_errno_adjtimex** function is used to obtain an explanation of an error returned by the *adjtimex*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *adjtimex*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = adjtimex(data);
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_adjtimex(message, sizeof(message), err,
```

```
                data);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_adjtimex_or_die*(3) function.

## SEE ALSO

*adjtimex*(2)
: tune kernel clock

*explain_adjtimex_or_die*(3)
: tune kernel clock and report errors

## COPYRIGHT

## NAME

explain_adjtimex_or_die − tune kernel clock and report errors

## SYNOPSIS

#include <libexplain/adjtimex.h>

int explain_adjtimex_or_die(struct timex *data);
int explain_adjtimex_on_error(struct timex *data);

## DESCRIPTION

The **explain_adjtimex_or_die** function is used to call the *adjtimex*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_adjtimex*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_adjtimex_on_error** function is used to call the *adjtimex*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_adjtimex*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *adjtimex*(2) system call.

## RETURN VALUE

The **explain_adjtimex_or_die** function only returns on success, see *adjtimex*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_adjtimex_on_error** function always returns the value return by the wrapped *adjtimex*(2) system call.

## EXAMPLE

The **explain_adjtimex_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_adjtimex_or_die(data);
```

## SEE ALSO

*adjtimex*(2)
tune kernel clock

*explain_adjtimex*(3)
explain *adjtimex*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_bind − explain bind(2) errors

## SYNOPSIS

#include <libexplain/bind.h>

const char *explain_bind(int fildes, const struct sockaddr *sock_addr, int sock_addr_size);
const char *explain_errno_bind(int errnum, int fildes, const struct sockaddr *sock_addr, int sock_addr_size);
void explain_message_bind(char *message, int message_size, int fildes, const struct sockaddr *sock_addr, int sock_addr_size);
void explain_message_errno_bind(char *message, int message_size, int errnum, int fildes, const struct sockaddr *sock_addr, int sock_addr_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *bind*(2) system call.

### explain_bind

const char *explain_bind(int fildes, const struct sockaddr *sock_addr, int sock_addr_size);

The **explain_bind** function is used to obtain an explanation of an error returned by the *bind*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (bind(fildes, sock_addr, sock_addr_size) < 0)
{
    fprintf(stderr, "%s\n",
        explain_bind(fildes, sock_addr, sock_addr_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_bind_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *bind*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *bind*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *bind*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_bind

const char *explain_errno_bind(int errnum, int fildes, const struct sockaddr *sock_addr, int sock_addr_size);

The **explain_errno_bind** function is used to obtain an explanation of an error returned by the *bind*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (bind(fildes, sock_addr, sock_addr_size) < 0)
{
    int err = errno;
```

```
        fprintf(stderr, "%s\n", explain_errno_bind(err,
            fildes, sock_addr, sock_addr_size));
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_bind_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *bind*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *bind*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *bind*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_bind**

void explain_message_bind(char *message, int message_size, int fildes, const struct sockaddr *sock_addr, int sock_addr_size);

The **explain_message_bind** function may be used to obtain an explanation of an error returned by the *bind*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
    if (bind(fildes, sock_addr, sock_addr_size) < 0)
    {
        char message[3000];
        explain_message_bind(message, sizeof(message),
            fildes, sock_addr, sock_addr_size);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_bind_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *bind*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *bind*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *bind*(2) system call.

**explain_message_errno_bind**

void explain_message_errno_bind(char *message, int message_size, int errnum, int fildes, const struct sockaddr *sock_addr, int sock_addr_size);

The **explain_message_errno_bind** function may be used to obtain an explanation of an error returned by the *bind*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (bind(fildes, sock_addr, sock_addr_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_bind(message, sizeof(message), err,
        fildes, sock_addr, sock_addr_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_bind_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *bind*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *bind*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *bind*(2) system call.

## SEE ALSO

*bind*(2) bind a name to a socket

*explain_bind_or_die*(3)
bind a name to a socket and report errors

## COPYRIGHT

## NAME

explain_bind_or_die − bind a name to a socket and report errors

## SYNOPSIS

#include <libexplain/bind.h>

void explain_bind_or_die(int fildes, const struct sockaddr *sock_addr, int sock_addr_size);

## DESCRIPTION

The **explain_bind_or_die** function is used to call the *bind*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_bind*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_bind_or_die(fildes, sock_addr, sock_addr_size);
```

*fildes* The fildes, exactly as to be passed to the *bind*(2) system call.

*sock_addr*
The sock_addr, exactly as to be passed to the *bind*(2) system call.

*sock_addr_size*
The sock_addr_size, exactly as to be passed to the *bind*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*bind*(2) bind a name to a socket

*explain_bind*(3)
explain *bind*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_calloc − explain *calloc*(3) errors

## SYNOPSIS

#include <libexplain/calloc.h>

const char *explain_calloc(size_t nmemb, size_t size);
const char *explain_errno_calloc(int errnum, size_t nmemb, size_t size);
void explain_message_calloc(char *message, int message_size, size_t nmemb, size_t size);
void explain_message_errno_calloc(char *message, int message_size, int errnum, size_t nmemb, size_t size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *calloc*(3) system call.

### explain_calloc

const char *explain_calloc(size_t nmemb, size_t size);

The **explain_calloc** function is used to obtain an explanation of an error returned by the *calloc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*nmemb* The original nmemb, exactly as passed to the *calloc*(3) system call.

*size* The original size, exactly as passed to the *calloc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void *result = calloc(nmemb, size);
if (!result && errno != 0)
{
   fprintf(stderr, "%s\n", explain_calloc(nmemb, size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_calloc_or_die*(3) function.

### explain_errno_calloc

const char *explain_errno_calloc(int errnum, size_t nmemb, size_t size);

The **explain_errno_calloc** function is used to obtain an explanation of an error returned by the *calloc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nmemb* The original nmemb, exactly as passed to the *calloc*(3) system call.

*size* The original size, exactly as passed to the *calloc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void *result = calloc(nmemb, size);
if (!result && errno != 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_calloc(err, nmemb,
   size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_calloc_or_die*(3) function.

**explain_message_calloc**

void explain_message_calloc(char *message, int message_size, size_t nmemb, size_t size);

The **explain_message_calloc** function is used to obtain an explanation of an error returned by the *calloc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nmemb* The original nmemb, exactly as passed to the *calloc*(3) system call.

*size* The original size, exactly as passed to the *calloc*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void *result = calloc(nmemb, size);
if (!result && errno != 0)
{
    char message[3000];
   explain_message_calloc(message, sizeof(message), nmemb, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_calloc_or_die*(3) function.

**explain_message_errno_calloc**

void explain_message_errno_calloc(char *message, int message_size, int errnum, size_t nmemb, size_t size);

The **explain_message_errno_calloc** function is used to obtain an explanation of an error returned by the *calloc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*nmemb* The original nmemb, exactly as passed to the *calloc*(3) system call.

*size* The original size, exactly as passed to the *calloc*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void *result = calloc(nmemb, size);
if (!result && errno != 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_calloc(message, sizeof(message), err,
   nmemb, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_calloc_or_die*(3) function.

## SEE ALSO

*calloc*(3)
Allocate and clear memory

*explain_calloc_or_die*(3)
Allocate and clear memory and report errors

## COPYRIGHT

## NAME

explain_calloc_or_die − Allocate and clear memory and report errors

## SYNOPSIS

#include <libexplain/calloc.h>

void *explain_calloc_or_die(size_t nmemb, size_t size);
void *explain_calloc_on_error(size_t nmemb, size_t size);

## DESCRIPTION

The **explain_calloc_or_die** function is used to call the *calloc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_calloc*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_calloc_on_error** function is used to call the *calloc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_calloc*(3) function, but still returns to the caller.

*nmemb* The nmemb, exactly as to be passed to the *calloc*(3) system call.

*size* The size, exactly as to be passed to the *calloc*(3) system call.

## RETURN VALUE

The **explain_calloc_or_die** function only returns on success, see *calloc*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_calloc_on_error** function always returns the value return by the wrapped *calloc*(3) system call.

## EXAMPLE

The **explain_calloc_or_die** function is intended to be used in a fashion similar to the following example:

```
void *result = explain_calloc_or_die(nmemb, size);
```

## SEE ALSO

*calloc*(3)
Allocate and clear memory

*explain_calloc*(3)
explain *calloc*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_chdir − explain chdir(2) errors

## SYNOPSIS

```
#include <libexplain/chdir.h>
const char *explain_chdir(const char *pathname);
void explain_message_chdir(char *message, int message_size, const char *pathname);
const char *explain_errno_chdir(int errnum, const char *pathname);
void explain_message_errno_chdir(char *message, int message_size, int errnum, const char *pathname);
```

## DESCRIPTION

These function may be used to obtain explanations of *chdir*(2) errors.

### explain_chdir

const char *explain_chdir(const char *pathname);

The explain_chdir function is used to obtain an explanation of an error returned by the *chdir*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (chdir(pathname) < 0)
{
    fprintf(stderr, '%s0, explain_chdir(pathname));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *chdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_chdir

const char *explain_errno_chdir(int errnum, const char *pathname);

The explain_errno_chdir function is used to obtain an explanation of an error returned by the *chdir*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (chdir(pathname) < 0)
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_chdir(err, pathname));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *chdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_chdir**

void explain_message_chdir(char *message, int message_size, const char *pathname);

The explain_message_chdir function is used to obtain an explanation of an error returned by the *chdir*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (chdir(pathname) < 0)
{
    char message[3000];
    explain_message_chdir(message, sizeof(message), pathname);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *chdir*(2) system call.

**explain_message_errno_chdir**

void explain_message_errno_chdir(char *message, int message_size, int errnum, const char * pathname);

The explain_message_errno_chdir function is used to obtain an explanation of an error returned by the *chdir*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (chdir(pathname) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_chdir(message, sizeof(message), err,
        pathname);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *chdir*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_chdir_or_die − change working directory and report errors

## SYNOPSIS

#include <libexplain/chdir.h>

void explain_chdir_or_die(const char * pathname);

## DESCRIPTION

The **explain_chdir_or_die** function is used to call the *chdir*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_chdir*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_chdir_or_die(pathname);
```

*pathname*

The pathname, exactly as to be passed to the *chdir*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## COPYRIGHT

## NAME

explain_chmod − explain chmod(2) errors

## SYNOPSIS

#include <libexplain/chmod.h>
const char *explain_chmod(const char *pathname, int mode);
const char *explain_errno_chmod(int errnum, const char *pathname, int mode);
void explain_message_chmod(char *message, int message_size, const char *pathname, int mode);
void explain_message_errno_chmod(char *message, int message_size, int errnum, const char *pathname, int mode);

## DESCRIPTION

These functions may be used to otain explanations for *chmod*(2) errors.

### explain_chmod

const char *explain_chmod(const char *pathname, int mode);

The explain_chmod function is used to obtain an explanation of an error returned by the *chmod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (chmod(pathname, mode) < 0)
{
    fprintf(stderr, '%s0, explain_chmod(pathname, mode));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *chmod*(2) system call.

*mode*
: The original mode, exactly as passed to the *chmod*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_chmod

const char *explain_errno_chmod(int errnum, const char *pathname, int mode);

The explain_errno_chmod function is used to obtain an explanation of an error returned by the *chmod*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (chmod(pathname) < 0)
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_chmod(err, pathname));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *chmod*(2) system call.

*mode* The original mode, exactly as passed to the *chmod*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_chmod**

void explain_message_chmod(char *message, int message_size, const char *pathname, int mode);

The explain_message_chmod function is used to obtain an explanation of an error returned by the *chmod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (chmod(pathname, mode) < 0)
{
    char message[3000];
    explain_message_chmod(message, sizeof(message), pathname, mode);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *chmod*(2) system call.

*mode* The original mode, exactly as passed to the *chmod*(2) system call.

**explain_message_errno_chmod**

void explain_message_errno_chmod(char * message, int message_size, int errnum, const char *pathname, int mode);

The explain_message_errno_chmod function is used to obtain an explanation of an error returned by the *chmod*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (chmod(pathname) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_chmod(message, sizeof(message), err,
        pathname);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *chmod*(2) system call.

*mode* The original mode, exactly as passed to the *chmod*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_chmod_or_die − change permissions of a file and report errors

## SYNOPSIS

#include <libexplain/chmod.h>
void explain_chmod_or_die(const char *pathname, int mode);

## DESCRIPTION

The explain_chmod_or_die function is used to call the *chmod*(2) system call. On failure an explanation wiil be printed to stderr, obtained from *explain_chmod*(3), and the the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_chmod_or_die(pathname, mode);
```

*pathname*
: The pathname, exactly as to be passed to the *chmod*(2) system call.

*mode*
: The mode, exactly as to be passed to the *chmod*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exit(EXIT_FAILURE)s.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_chown − explain chown(2) errors

## SYNOPSIS

#include <libexplain/chown.h>

const char *explain_chown(const char *pathname, int owner, int group);
const char *explain_errno_chown(int errnum, const char *pathname, int owner, int group);
void explain_message_chown(char *message, int message_size, const char *pathname, int owner, int group);
void explain_message_errno_chown(char *message, int message_size, int errnum, const char *pathname, int owner, int group);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *chown*(2) system call.

### explain_chown

const char *explain_chown(const char *pathname, int owner, int group);

The **explain_chown** function is used to obtain an explanation of an error returned by the *chown*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (chown(pathname, owner, group) < 0)
{
    fprintf(stderr, "%s\n", explain_chown(pathname, owner, group));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *chown*(2) system call.

*owner*
: The original owner, exactly as passed to the *chown*(2) system call.

*group*
: The original group, exactly as passed to the *chown*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_chown

const char *explain_errno_chown(int errnum, const char *pathname, int owner, int group);

The **explain_errno_chown** function is used to obtain an explanation of an error returned by the *chown*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (chown(pathname, owner, group) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_chown(err, pathname, owner,
        group));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *chown*(2) system call.

*owner* The original owner, exactly as passed to the *chown*(2) system call.

*group* The original group, exactly as passed to the *chown*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_chown**

void explain_message_chown(char *message, int message_size, const char *pathname, int owner, int group);

The **explain_message_chown** function may be used to obtain an explanation of an error returned by the *chown*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (chown(pathname, owner, group) < 0)
{
    char message[3000];
    explain_message_chown(message, sizeof(message), pathname, owner, group);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *chown*(2) system call.

*owner* The original owner, exactly as passed to the *chown*(2) system call.

*group* The original group, exactly as passed to the *chown*(2) system call.

**explain_message_errno_chown**

void explain_message_errno_chown(char *message, int message_size, int errnum, const char *pathname, int owner, int group);

The **explain_message_errno_chown** function may be used to obtain an explanation of an error returned by the *chown*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (chown(pathname, owner, group) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_chown(message, sizeof(message), err,
        pathname, owner, group);
```

```
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *chown*(2) system call.

*owner* The original owner, exactly as passed to the *chown*(2) system call.

*group* The original group, exactly as passed to the *chown*(2) system call.

## SEE ALSO

*chown*(2)
change ownership of a file

*explain_chown_or_die*(3)
change ownership of a file and report errors

## COPYRIGHT

## NAME

explain_chown_or_die − change ownership of a file and report errors

## SYNOPSIS

#include <libexplain/chown.h>

void explain_chown_or_die(const char *pathname, int owner, int group);

## DESCRIPTION

The **explain_chown_or_die** function is used to call the *chown*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_chown*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_chown_or_die(pathname, owner, group);
```

*pathname*
: The pathname, exactly as to be passed to the *chown*(2) system call.

*owner*
: The owner, exactly as to be passed to the *chown*(2) system call.

*group*
: The group, exactly as to be passed to the *chown*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*chown*(2)
: change ownership of a file

*explain_chown*(3)
: explain *chown*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_chroot − explain chroot(2) errors

## SYNOPSIS

#include <libexplain/chroot.h>

const char *explain_chroot(const char *pathname);
const char *explain_errno_chroot(int errnum, const char *pathname);
void explain_message_chroot(char *message, int message_size, const char *pathname);
void explain_message_errno_chroot(char *message, int message_size, int errnum, const char *pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *chroot*(2) system call.

### explain_chroot

const char *explain_chroot(const char *pathname);

The **explain_chroot** function is used to obtain an explanation of an error returned by the *chroot*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *chroot*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (chroot(pathname) < 0)
{
   fprintf(stderr, "%s\n", explain_chroot(pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_chroot_or_die*(3) function.

### explain_errno_chroot

const char *explain_errno_chroot(int errnum, const char *pathname);

The **explain_errno_chroot** function is used to obtain an explanation of an error returned by the *chroot*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *chroot*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (chroot(pathname) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_chroot(err, pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_chroot_or_die*(3) function.

**explain_message_chroot**

void explain_message_chroot(char *message, int message_size, const char *pathname);

The **explain_message_chroot** function is used to obtain an explanation of an error returned by the *chroot*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *chroot*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (chroot(pathname) < 0)
{
    char message[3000];
    explain_message_chroot(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_chroot_or_die*(3) function.

**explain_message_errno_chroot**

void explain_message_errno_chroot(char *message, int message_size, int errnum, const char *pathname);

The **explain_message_errno_chroot** function is used to obtain an explanation of an error returned by the *chroot*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *chroot*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (chroot(pathname) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_chroot(message, sizeof(message), err,
```

```
                pathname);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_chroot_or_die*(3) function.

## SEE ALSO

*chroot*(2)
change root directory

*explain_chroot_or_die*(3)
change root directory and report errors

## COPYRIGHT

## NAME

explain_chroot_or_die − change root directory and report errors

## SYNOPSIS

#include <libexplain/chroot.h>

void explain_chroot_or_die(const char *pathname);
int explain_chroot_on_error(const char *pathname))

## DESCRIPTION

The **explain_chroot_or_die** function is used to call the *chroot*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_chroot*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_chroot_on_error** function is used to call the *chroot*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_chroot*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *chroot*(2) system call.

## RETURN VALUE

The **explain_chroot_or_die** function only returns on success, see *chroot*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_chroot_on_error** function always returns the value return by the wrapped *chroot*(2) system call.

## EXAMPLE

The **explain_chroot_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_chroot_or_die(pathname);
```

## SEE ALSO

*chroot*(2)
: change root directory

*explain_chroot*(3)
: explain *chroot*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_close − explain close(2) errors

## SYNOPSIS

#include <libexplain/close.h>

const char *explain_close(int fildes);
const char *explain_errno_close(int errnum, int fildes);
void explain_message_close(char *message, int message_size, int fildes);
void explain_message_errno_close(char *message, int message_size, int errnum, int fildes);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *close*(2) system call.

### explain_close

const char *explain_close(int fildes);

The **explain_close** function is used to obtain an explanation of an error returned by the *close*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (close(fildes) < 0)
{
    fprintf(stderr, "%s\n", explain_close(fildes));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *close*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_close

const char *explain_errno_close(int errnum, int fildes);

The **explain_errno_close** function is used to obtain an explanation of an error returned by the *close*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (close(fildes) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_close(err, fildes));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *close*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_close**

void explain_message_close(char *message, int message_size, int fildes);

The **explain_message_close** function is used to obtain an explanation of an error returned by the *close*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (close(fildes) < 0)
{
    char message[3000];
    explain_message_close(message, sizeof(message), fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *close*(2) system call.

**explain_message_errno_close**

void explain_message_errno_close(char *message, int message_size, int errnum, int fildes);

The **explain_message_errno_close** function is used to obtain an explanation of an error returned by the *close*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (close(fildes) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_close(message, sizeof(message), err, fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *close*(2) system call.

## SEE ALSO

*close* close a file descriptor

*explain_close_or_die*
close a file descriptor and report errors

## COPYRIGHT

## NAME

explain_closedir − explain closedir(3) errors

## SYNOPSIS

#include <libexplain/closedir.h>

const char *explain_closedir(DIR *dir);
const char *explain_errno_closedir(int errnum, DIR *dir);
void explain_message_closedir(char *message, int message_size, DIR *dir);
void explain_message_errno_closedir(char *message, int message_size, int errnum, DIR *dir);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *closedir*(3) system call.

### explain_closedir

const char *explain_closedir(DIR *dir);

The **explain_closedir** function is used to obtain an explanation of an error returned by the *closedir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (closedir(dir) < 0)
{
    fprintf(stderr, "%s\n", explain_closedir(dir));
    exit(EXIT_FAILURE);
}
```

*dir* The original dir, exactly as passed to the *closedir*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_closedir

const char *explain_errno_closedir(int errnum, DIR *dir);

The **explain_errno_closedir** function is used to obtain an explanation of an error returned by the *closedir*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (closedir(dir) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_closedir(err, dir));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *closedir*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_closedir**

void explain_message_closedir(char *message, int message_size, DIR *dir);

The **explain_message_closedir** function may be used to obtain an explanation of an error returned by the *closedir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (closedir(dir) < 0)
{
    char message[3000];
    explain_message_closedir(message, sizeof(message), dir);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*dir* The original dir, exactly as passed to the *closedir*(3) system call.

**explain_message_errno_closedir**

void explain_message_errno_closedir(char *message, int message_size, int errnum, DIR *dir);

The **explain_message_errno_closedir** function may be used to obtain an explanation of an error returned by the *closedir*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (closedir(dir) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_closedir(message, sizeof(message), err, dir);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *closedir*(3) system call.

## SEE ALSO

*closedir*(3)
close a directory

*explain_closedir_or_die*(3)
close a directory and report errors

## COPYRIGHT

## NAME

explain_closedir_or_die − close a directory and report errors

## SYNOPSIS

#include <libexplain/closedir.h>

void explain_closedir_or_die(DIR *dir);

## DESCRIPTION

The **explain_closedir_or_die** function is used to call the *closedir*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_closedir*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_closedir_or_die(dir);
```

*dir* The dir, exactly as to be passed to the *closedir*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*closedir*(3)
close a directory

*explain_closedir*(3)
explain *closedir*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_close_or_die − close a file descriptor and report errors

## SYNOPSIS

#include <libexplain/close.h>

void explain_close_or_die(int fildes);

## DESCRIPTION

The **explain_close_or_die** function is used to call the *close*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_close*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_close_or_die(fildes);
```

*fildes* The fildes, exactly as to be passed to the *close*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*close*(2) close a file descriptor

*explain_close*(3)
explain *close*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_connect − explain connect(2) errors

## SYNOPSIS

#include <libexplain/connect.h>

const char *explain_connect(int fildes, const struct sockaddr *serv_addr, int serv_addr_size);
const char *explain_errno_connect(int errnum, int fildes, const struct sockaddr *serv_addr, int serv_addr_size);
void explain_message_connect(char *message, int message_size, int fildes, const struct sockaddr *serv_addr, int serv_addr_size);
void explain_message_errno_connect(char *message, int message_size, int errnum, int fildes, const struct sockaddr *serv_addr, int serv_addr_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *connect*(2) system call.

### explain_connect

const char *explain_connect(int fildes, const struct sockaddr *serv_addr, int serv_addr_size);

The **explain_connect** function is used to obtain an explanation of an error returned by the *connect*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (connect(fildes, serv_addr, serv_addr_size) < 0)
{
    fprintf(stderr, "%s\n", explain_connect(fildes, serv_addr,
        serv_addr_size));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *connect*(2) system call.

*serv_addr*
The original serv_addr, exactly as passed to the *connect*(2) system call.

*serv_addr_size*
The original serv_addr_size, exactly as passed to the *connect*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_connect

const char *explain_errno_connect(int errnum, int fildes, const struct sockaddr *serv_addr, int serv_addr_size);

The **explain_errno_connect** function is used to obtain an explanation of an error returned by the *connect*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (connect(fildes, serv_addr, serv_addr_size) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_connect(err,
        fildes, serv_addr, serv_addr_size));
```

```
        exit(EXIT_FAILURE);
    }
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *connect*(2) system call.

*serv_addr*
The original serv_addr, exactly as passed to the *connect*(2) system call.

*serv_addr_size*
The original serv_addr_size, exactly as passed to the *connect*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_connect**

void explain_message_connect(char *message, int message_size, int fildes, const struct sockaddr *serv_addr, int serv_addr_size);

The **explain_message_connect** function may be used to obtain an explanation of an error returned by the *connect*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
    if (connect(fildes, serv_addr, serv_addr_size) < 0)
    {
        char message[3000];
        explain_message_connect(message, sizeof(message),
            fildes, serv_addr, serv_addr_size);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *connect*(2) system call.

*serv_addr*
The original serv_addr, exactly as passed to the *connect*(2) system call.

*serv_addr_size*
The original serv_addr_size, exactly as passed to the *connect*(2) system call.

**explain_message_errno_connect**

void explain_message_errno_connect(char *message, int message_size, int errnum, int fildes, const struct sockaddr *serv_addr, int serv_addr_size);

The **explain_message_errno_connect** function may be used to obtain an explanation of an error returned by the *connect*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (connect(fildes, serv_addr, serv_addr_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_connect(message, sizeof(message), err,
        fildes, serv_addr, serv_addr_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *connect*(2) system call.

*serv_addr*
The original serv_addr, exactly as passed to the *connect*(2) system call.

*serv_addr_size*
The original serv_addr_size, exactly as passed to the *connect*(2) system call.

## SEE ALSO

*connect*(2)
initiate a connection on a socket

*explain_connect_or_die*(3)
initiate a connection on a socket and report errors

## COPYRIGHT

## NAME

explain_connect_or_die − initiate a connection on a socket and report errors

## SYNOPSIS

#include <libexplain/connect.h>

void explain_connect_or_die(int fildes, const struct sockaddr *serv_addr, int serv_addr_size);

## DESCRIPTION

The **explain_connect_or_die** function is used to call the *connect*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_connect*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_connect_or_die(fildes, serv_addr, serv_addr_size);
```

*fildes* The fildes, exactly as to be passed to the *connect*(2) system call.

*serv_addr*
: The serv_addr, exactly as to be passed to the *connect*(2) system call.

*serv_addr_size*
: The serv_addr_size, exactly as to be passed to the *connect*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*connect*(2)
: initiate a connection on a socket

*explain_connect*(3)
: explain *connect*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_creat − explain creat(2) errors

## SYNOPSIS

#include <libexplain/creat.h>

const char *explain_creat(const char *pathname, int mode);
const char *explain_errno_creat(int errnum, const char *pathname, int mode);
void explain_message_creat(char *message, int message_size, const char *pathname, int mode);
void explain_message_errno_creat(char *message, int message_size, int errnum, const char *pathname, int mode);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *creat*(2) system call.

### explain_creat

const char *explain_creat(const char *pathname, int mode);

The **explain_creat** function is used to obtain an explanation of an error returned by the *creat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (creat(pathname, mode) < 0)
{
    fprintf(stderr, "%s\n", explain_creat(pathname, mode));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *creat*(2) system call.

*mode*
: The original mode, exactly as passed to the *creat*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_creat

const char *explain_errno_creat(int errnum, const char *pathname, int mode);

The **explain_errno_creat** function is used to obtain an explanation of an error returned by the *creat*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (creat(pathname, mode) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_creat(err, pathname, mode));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *creat*(2) system call.

*mode* The original mode, exactly as passed to the *creat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_creat**

void explain_message_creat(char *message, int message_size, const char *pathname, int mode);

The **explain_message_creat** function may be used to obtain an explanation of an error returned by the *creat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (creat(pathname, mode) < 0)
{
    char message[3000];
    explain_message_creat(message, sizeof(message), pathname, mode);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *creat*(2) system call.

*mode* The original mode, exactly as passed to the *creat*(2) system call.

**explain_message_errno_creat**

void explain_message_errno_creat(char *message, int message_size, int errnum, const char *pathname, int mode);

The **explain_message_errno_creat** function may be used to obtain an explanation of an error returned by the *creat*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (creat(pathname, mode) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_creat(message, sizeof(message), err, pathname,
        mode);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *creat*(2) system call.

*mode* The original mode, exactly as passed to the *creat*(2) system call.

## SEE ALSO

*creat*(2) open and possibly create a file or device

*explain_creat_or_die*(3)
create and open a file and report errors

## COPYRIGHT

## NAME

explain_creat_or_die − create and open a file creat and report errors

## SYNOPSIS

#include <libexplain/creat.h>

void explain_creat_or_die(const char *pathname, int mode);

## DESCRIPTION

The **explain_creat_or_die** function is used to call the *creat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_creat*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_creat_or_die(pathname, mode);
```

*pathname*
: The pathname, exactly as to be passed to the *creat*(2) system call.

*mode*
: The mode, exactly as to be passed to the *creat*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*creat*(2) open and possibly create a file or device

*explain_creat*(3)
: explain *creat*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_dirfd − explain dirfd(3) errors

## SYNOPSIS

#include <libexplain/dirfd.h>

const char *explain_dirfd(DIR *dir);
const char *explain_errno_dirfd(int errnum, DIR *dir);
void explain_message_dirfd(char *message, int message_size, DIR *dir);
void explain_message_errno_dirfd(char *message, int message_size, int errnum, DIR *dir);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *dirfd*(3) system call.

### explain_dirfd

const char *explain_dirfd(DIR *dir);

The **explain_dirfd** function is used to obtain an explanation of an error returned by the *dirfd*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*dir* The original dir, exactly as passed to the *dirfd*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = dirfd(dir);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_dirfd(dir));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_dirfd_or_die*(3) function.

### explain_errno_dirfd

const char *explain_errno_dirfd(int errnum, DIR *dir);

The **explain_errno_dirfd** function is used to obtain an explanation of an error returned by the *dirfd*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *dirfd*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = dirfd(dir);
```

```
if (result < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_dirfd(err, dir));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_dirfd_or_die*(3) function.

**explain_message_dirfd**

void explain_message_dirfd(char *message, int message_size, DIR *dir);

The **explain_message_dirfd** function is used to obtain an explanation of an error returned by the *dirfd*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*dir* The original dir, exactly as passed to the *dirfd*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = dirfd(dir);
if (result < 0)
{
    char message[3000];
    explain_message_dirfd(message, sizeof(message), dir);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_dirfd_or_die*(3) function.

**explain_message_errno_dirfd**

void explain_message_errno_dirfd(char *message, int message_size, int errnum, DIR *dir);

The **explain_message_errno_dirfd** function is used to obtain an explanation of an error returned by the *dirfd*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *dirfd*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = dirfd(dir);
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_dirfd(message, sizeof(message), err,
```

```
                dir);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_dirfd_or_die*(3) function.

## SEE ALSO

*dirfd*(3) get directory stream file descriptor

*explain_dirfd_or_die*(3)
get directory stream file descriptor and report errors

## COPYRIGHT

## NAME

explain_dirfd_or_die − get directory stream file descriptor and report errors

## SYNOPSIS

#include <libexplain/dirfd.h>

int explain_dirfd_or_die(DIR *dir);
int explain_dirfd_on_error(DIR *dir);

## DESCRIPTION

The **explain_dirfd_or_die** function is used to call the *dirfd*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_dirfd*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_dirfd_on_error** function is used to call the *dirfd*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_dirfd*(3) function, but still returns to the caller.

*dir* The dir, exactly as to be passed to the *dirfd*(3) system call.

## RETURN VALUE

The **explain_dirfd_or_die** function only returns on success, see *dirfd*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_dirfd_on_error** function always returns the value return by the wrapped *dirfd*(3) system call.

## EXAMPLE

The **explain_dirfd_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_dirfd_or_die(dir);
```

## SEE ALSO

*dirfd*(3) get directory stream file descriptor

*explain_dirfd*(3)
explain *dirfd*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_dup2 − explain dup2(2) errors

## SYNOPSIS

#include <libexplain/dup2.h>

const char *explain_dup2(int oldfd, int newfd);
const char *explain_errno_dup2(int errnum, int oldfd, int newfd);
void explain_message_dup2(char *message, int message_size, int oldfd, int newfd);
void explain_message_errno_dup2(char *message, int message_size, int errnum, int oldfd, int newfd);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *dup2*(2) system call.

### explain_dup2

const char *explain_dup2(int oldfd, int newfd);

The **explain_dup2** function is used to obtain an explanation of an error returned by the *dup2*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (dup2(oldfd, newfd) < 0)
{
    fprintf(stderr, "%s\n", explain_dup2(oldfd, newfd));
    exit(EXIT_FAILURE);
}
```

*oldfd* The original oldfd, exactly as passed to the *dup2*(2) system call.

*newfd* The original newfd, exactly as passed to the *dup2*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_dup2

const char *explain_errno_dup2(int errnum, int oldfd, int newfd);

The **explain_errno_dup2** function is used to obtain an explanation of an error returned by the *dup2*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (dup2(oldfd, newfd) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_dup2(err, oldfd, newfd));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*oldfd* The original oldfd, exactly as passed to the *dup2*(2) system call.

*newfd* The original newfd, exactly as passed to the *dup2*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_dup2**

void explain_message_dup2(char *message, int message_size, int oldfd, int newfd);

The **explain_message_dup2** function may be used to obtain an explanation of an error returned by the *dup2*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (dup2(oldfd, newfd) < 0)
{
    char message[3000];
    explain_message_dup2(message, sizeof(message), oldfd, newfd);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*oldfd* The original oldfd, exactly as passed to the *dup2*(2) system call.

*newfd* The original newfd, exactly as passed to the *dup2*(2) system call.

**explain_message_errno_dup2**

void explain_message_errno_dup2(char *message, int message_size, int errnum, int oldfd, int newfd);

The **explain_message_errno_dup2** function may be used to obtain an explanation of an error returned by the *dup2*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (dup2(oldfd, newfd) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_dup2(message, sizeof(message), err, oldfd, newfd);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*oldfd* The original oldfd, exactly as passed to the *dup2*(2) system call.

*newfd* The original newfd, exactly as passed to the *dup2*(2) system call.

## SEE ALSO

*dup2*(2) duplicate a file descriptor

*explain_dup2_or_die*(3)
duplicate a file descriptor and report errors

## COPYRIGHT

## NAME

explain_dup2_or_die − duplicate a file descriptor and report errors

## SYNOPSIS

#include <libexplain/dup2.h>

void explain_dup2_or_die(int oldfd, int newfd);

## DESCRIPTION

The **explain_dup2_or_die** function is used to call the *dup2*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_dup2*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_dup2_or_die(oldfd, newfd);
```

*oldfd* The oldfd, exactly as to be passed to the *dup2*(2) system call.

*newfd* The newfd, exactly as to be passed to the *dup2*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*dup2*(2) duplicate a file descriptor

*explain_dup2*(3)
explain *dup2*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_dup − explain dup(2) errors

## SYNOPSIS

#include <libexplain/dup.h>

const char *explain_dup(int fildes);
const char *explain_errno_dup(int errnum, int fildes);
void explain_message_dup(char *message, int message_size, int fildes);
void explain_message_errno_dup(char *message, int message_size, int errnum, int fildes);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *dup*(2) system call.

### explain_dup

const char *explain_dup(int fildes);

The **explain_dup** function is used to obtain an explanation of an error returned by the *dup*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (dup(fildes) < 0)
{
    fprintf(stderr, "%s\n", explain_dup(fildes));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *dup*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_dup

const char *explain_errno_dup(int errnum, int fildes);

The **explain_errno_dup** function is used to obtain an explanation of an error returned by the *dup*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (dup(fildes) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_dup(err, fildes));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *dup*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_dup**

void explain_message_dup(char *message, int message_size, int fildes);

The **explain_message_dup** function may be used to obtain an explanation of an error returned by the *dup*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (dup(fildes) < 0)
{
    char message[3000];
    explain_message_dup(message, sizeof(message), fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *dup*(2) system call.

**explain_message_errno_dup**

void explain_message_errno_dup(char *message, int message_size, int errnum, int fildes);

The **explain_message_errno_dup** function may be used to obtain an explanation of an error returned by the *dup*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (dup(fildes) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_dup(message, sizeof(message), err, fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *dup*(2) system call.

## SEE ALSO

*dup*(2) duplicate a file descriptor

*explain_dup_or_die*(3)
duplicate a file descriptor and report errors

## COPYRIGHT

## NAME

explain_dup_or_die − duplicate a file descriptor and report errors

## SYNOPSIS

#include <libexplain/dup.h>

void explain_dup_or_die(int fildes);

## DESCRIPTION

The **explain_dup_or_die** function is used to call the *dup*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_dup*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_dup_or_die(fildes);
```

*fildes* The fildes, exactly as to be passed to the *dup*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*dup*(2) duplicate a file descriptor

*explain_dup*(3)
explain *dup*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_eventfd − explain eventfd(2) errors

## SYNOPSIS

#include <libexplain/eventfd.h>

const char *explain_eventfd(unsigned int initval, int flags);
const char *explain_errno_eventfd(int errnum, unsigned int initval, int flags);
void explain_message_eventfd(char *message, int message_size, unsigned int initval, int flags);
void explain_message_errno_eventfd(char *message, int message_size, int errnum, unsigned int initval, int flags);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *eventfd*(2) system call.

### explain_eventfd

const char *explain_eventfd(unsigned int initval, int flags);

The **explain_eventfd** function is used to obtain an explanation of an error returned by the *eventfd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*initval* The original initval, exactly as passed to the *eventfd*(2) system call.

*flags* The original flags, exactly as passed to the *eventfd*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = eventfd(initval, flags);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_eventfd(initval, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_eventfd_or_die*(3) function.

### explain_errno_eventfd

const char *explain_errno_eventfd(int errnum, unsigned int initval, int flags);

The **explain_errno_eventfd** function is used to obtain an explanation of an error returned by the *eventfd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*initval* The original initval, exactly as passed to the *eventfd*(2) system call.

*flags* The original flags, exactly as passed to the *eventfd*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = eventfd(initval, flags);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_eventfd(err, initval,
   flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_eventfd_or_die*(3) function.

**explain_message_eventfd**

void explain_message_eventfd(char *message, int message_size, unsigned int initval, int flags);

The **explain_message_eventfd** function is used to obtain an explanation of an error returned by the *eventfd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*initval* The original initval, exactly as passed to the *eventfd*(2) system call.

*flags* The original flags, exactly as passed to the *eventfd*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = eventfd(initval, flags);
if (result < 0)
{
    char message[3000];
   explain_message_eventfd(message, sizeof(message), initval,
   flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_eventfd_or_die*(3) function.

**explain_message_errno_eventfd**

void explain_message_errno_eventfd(char *message, int message_size, int errnum, unsigned int initval, int flags);

The **explain_message_errno_eventfd** function is used to obtain an explanation of an error returned by the *eventfd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*initval* The original initval, exactly as passed to the *eventfd*(2) system call.

*flags* The original flags, exactly as passed to the *eventfd*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = eventfd(initval, flags);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_eventfd(message, sizeof(message), err,
   initval, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_eventfd_or_die*(3) function.

## SEE ALSO

*eventfd*(2)
create a file descriptor for event notification

*explain_eventfd_or_die*(3)
create a file descriptor for event notification and report errors

## COPYRIGHT

## NAME

explain_eventfd_or_die − create event notify file descriptor and report errors

## SYNOPSIS

#include <libexplain/eventfd.h>

int explain_eventfd_or_die(unsigned int initval, int flags);
int explain_eventfd_on_error(unsigned int initval, int flags);

## DESCRIPTION

The **explain_eventfd_or_die** function is used to call the *eventfd*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_eventfd*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_eventfd_on_error** function is used to call the *eventfd*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_eventfd*(3) function, but still returns to the caller.

*initval* The initval, exactly as to be passed to the *eventfd*(2) system call.

*flags* The flags, exactly as to be passed to the *eventfd*(2) system call.

## RETURN VALUE

The **explain_eventfd_or_die** function only returns on success, see *eventfd*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_eventfd_on_error** function always returns the value return by the wrapped *eventfd*(2) system call.

## EXAMPLE

The **explain_eventfd_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_eventfd_or_die(initval, flags);
```

## SEE ALSO

*eventfd*(2)
create a file descriptor for event notification

*explain_eventfd*(3)
explain *eventfd*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_execlp − explain *execlp*(3) errors

## SYNOPSIS

#include <libexplain/execlp.h>

const char *explain_execlp(, ...);
const char *explain_errno_execlp(int errnum, , ...);
void explain_message_execlp(char *message, int message_size, , ...);
void explain_message_errno_execlp(char *message, int message_size, int errnum, , ...);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *execlp*(3) system call.

### explain_execlp

const char *explain_execlp(, ...);

The **explain_execlp** function is used to obtain an explanation of an error returned by the *execlp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (execlp() < 0)
{
    fprintf(stderr, "%s\n", explain_execlp());
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_execlp_or_die*(3) function.

### explain_errno_execlp

const char *explain_errno_execlp(int errnum, , ...);

The **explain_errno_execlp** function is used to obtain an explanation of an error returned by the *execlp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (execlp() < 0)
{
     int err = errno;
    fprintf(stderr, "%s\n", explain_errno_execlp(err, ));
     exit(EXIT_FAILURE);
```

```
}
```

The above code example is available pre-packaged as the *explain_execlp_or_die*(3) function.

**explain_message_execlp**

void explain_message_execlp(char *message, int message_size, , ...);

The **explain_message_execlp** function is used to obtain an explanation of an error returned by the *execlp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (execlp() < 0)
{
    char message[3000];
   explain_message_execlp(message, sizeof(message), );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_execlp_or_die*(3) function.

**explain_message_errno_execlp**

void explain_message_errno_execlp(char *message, int message_size, int errnum, , ...);

The **explain_message_errno_execlp** function is used to obtain an explanation of an error returned by the *execlp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (execlp() < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_execlp(message, sizeof(message), err, );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_execlp_or_die*(3) function.

## SEE ALSO

*execlp*(3)
execute a file

*explain_execlp_or_die*(3)
execute a file and report errors

## COPYRIGHT

## NAME

explain_execlp_or_die − execute a file and report errors

## SYNOPSIS

#include <libexplain/execlp.h>

void explain_execlp_or_die(, ...);
int explain_execlp_on_error(, ...);

## DESCRIPTION

The **explain_execlp_or_die** function is used to call the *execlp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_execlp*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_execlp_on_error** function is used to call the *execlp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_execlp*(3) function, but still returns to the caller.

## RETURN VALUE

The **explain_execlp_or_die** function only returns on success, see *execlp*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_execlp_on_error** function always returns the value return by the wrapped *execlp*(3) system call.

## EXAMPLE

The **explain_execlp_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_execlp_or_die();
```

## SEE ALSO

*execlp*(3)
execute a file

*explain_execlp*(3)
explain *execlp*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_execv − explain *execv*(3) errors

## SYNOPSIS

#include <libexplain/execv.h>

const char *explain_execv(const char *pathname, char *const*argv);
const char *explain_errno_execv(int errnum, const char *pathname, char *const*argv);
void explain_message_execv(char *message, int message_size, const char *pathname, char *const*argv);
void explain_message_errno_execv(char *message, int message_size, int errnum, const char *pathname, char *const*argv);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *execv*(3) system call.

### explain_execv

const char *explain_execv(const char *pathname, char *const*argv);

The **explain_execv** function is used to obtain an explanation of an error returned by the *execv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *execv*(3) system call.

*argv*
: The original argv, exactly as passed to the *execv*(3) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (execv(pathname, argv) < 0)
{
   fprintf(stderr, "%s\n", explain_execv(pathname, argv));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_execv_or_die*(3) function.

### explain_errno_execv

const char *explain_errno_execv(int errnum, const char *pathname, char *const*argv);

The **explain_errno_execv** function is used to obtain an explanation of an error returned by the *execv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *execv*(3) system call.

*argv*
: The original argv, exactly as passed to the *execv*(3) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (execv(pathname, argv) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_execv(err, pathname,
   argv));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_execv_or_die*(3) function.

**explain_message_execv**

void explain_message_execv(char *message, int message_size, const char *pathname, char *const*argv);

The **explain_message_execv** function is used to obtain an explanation of an error returned by the *execv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *execv*(3) system call.

*argv* The original argv, exactly as passed to the *execv*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (execv(pathname, argv) < 0)
{
    char message[3000];
   explain_message_execv(message, sizeof(message), pathname,
   argv);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_execv_or_die*(3) function.

**explain_message_errno_execv**

void explain_message_errno_execv(char *message, int message_size, int errnum, const char *pathname, char *const*argv);

The **explain_message_errno_execv** function is used to obtain an explanation of an error returned by the *execv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *execv*(3) system call.

*argv* The original argv, exactly as passed to the *execv*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (execv(pathname, argv) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_execv(message, sizeof(message), err,
   pathname, argv);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_execv_or_die*(3) function.

## SEE ALSO

*execv*(3) execute a file

*explain_execv_or_die*(3)
execute a file and report errors

## COPYRIGHT

## NAME

explain_execve − explain execve(2) errors

## SYNOPSIS

#include <libexplain/execve.h>

const char *explain_execve(const char *pathname, const char *const *argv, const char *const *envp);
const char *explain_errno_execve(int errnum, const char *pathname, const char *const *argv, const char *const *envp);
void explain_message_execve(char *message, int message_size, const char *pathname, const char *const *argv, const char *const *envp);
void explain_message_errno_execve(char *message, int message_size, int errnum, const char *pathname, const char *const *argv, const char *const *envp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *execve*(2) system call.

### explain_execve

const char *explain_execve(const char *pathname, const char *const *argv, const char *const *envp);

The **explain_execve** function is used to obtain an explanation of an error returned by the *execve*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
execve(pathname, argv, envp);
fprintf(stderr, "%s\n", explain_execve(pathname, argv, envp));
exit(EXIT_FAILURE);
```

*pathname*
: The original pathname, exactly as passed to the *execve*(2) system call.

*argv*
: The original argv, exactly as passed to the *execve*(2) system call.

*envp*
: The original envp, exactly as passed to the *execve*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_execve

const char *explain_errno_execve(int errnum, const char *pathname, const char *const *argv, const char *const *envp);

The **explain_errno_execve** function is used to obtain an explanation of an error returned by the *execve*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
execve(pathname, argv, envp);
int err = errno;
fprintf(stderr, "%s\n", explain_errno_execve(err, pathname, argv, envp));
exit(EXIT_FAILURE);
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *execve*(2) system call.

*argv* The original argv, exactly as passed to the *execve*(2) system call.

*envp* The original envp, exactly as passed to the *execve*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_execve**

void explain_message_execve(char *message, int message_size, const char *pathname, const char *const *argv, const char *const *envp);

The **explain_message_execve** function may be used to obtain an explanation of an error returned by the *execve*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
execve(pathname, argv, envp);
char message[3000];
explain_message_execve(message, sizeof(message), pathname, argv, envp);
fprintf(stderr, "%s\n", message);
exit(EXIT_FAILURE);
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *execve*(2) system call.

*argv* The original argv, exactly as passed to the *execve*(2) system call.

*envp* The original envp, exactly as passed to the *execve*(2) system call.

**explain_message_errno_execve**

void explain_message_errno_execve(char *message, int message_size, int errnum, const char *pathname, const char *const *argv, const char *const *envp);

The **explain_message_errno_execve** function may be used to obtain an explanation of an error returned by the *execve*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
execve(pathname, argv, envp);
int err = errno;
char message[3000];
explain_message_errno_execve(message, sizeof(message), err,
    pathname, argv, envp);
fprintf(stderr, "%s\n", message);
exit(EXIT_FAILURE);
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *execve*(2) system call.

*argv* The original argv, exactly as passed to the *execve*(2) system call.

*envp* The original envp, exactly as passed to the *execve*(2) system call.

## SEE ALSO

*execve*(2)
execute program

*explain_execve_or_die*(3)
execute program and report errors

## COPYRIGHT

## NAME

explain_execve_or_die − execute program and report errors

## SYNOPSIS

#include <libexplain/execve.h>

void explain_execve_or_die(const char *pathname, const char *const *argv, const char *const *envp);

## DESCRIPTION

The **explain_execve_or_die** function is used to call the *execve*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_execve*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_execve_or_die(pathname, argv, envp);
```

*pathname*
: The pathname, exactly as to be passed to the *execve*(2) system call.

*argv*
: The argv, exactly as to be passed to the *execve*(2) system call.

*envp*
: The envp, exactly as to be passed to the *execve*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*execve*(2)
: execute program

*explain_execve*(3)
: explain *execve*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_execv_or_die − execute a file and report errors

## SYNOPSIS

#include <libexplain/execv.h>

void explain_execv_or_die(const char *pathname, char *const*argv);
int explain_execv_on_error(const char *pathname, char *const*argv);

## DESCRIPTION

The **explain_execv_or_die** function is used to call the *execv*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_execv*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_execv_on_error** function is used to call the *execv*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_execv*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *execv*(3) system call.

*argv*
: The argv, exactly as to be passed to the *execv*(3) system call.

## RETURN VALUE

The **explain_execv_or_die** function only returns on success, see *execv*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_execv_on_error** function always returns the value return by the wrapped *execv*(3) system call.

## EXAMPLE

The **explain_execv_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_execv_or_die(pathname, argv);
```

## SEE ALSO

*execv*(3) execute a file

*explain_execv*(3)
: explain *execv*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_execvp − explain execvp(3) errors

## SYNOPSIS

#include <libexplain/execvp.h>

const char *explain_execvp(const char *pathname, char *const *argv);
const char *explain_errno_execvp(int errnum, const char *pathname, char *const *argv);
void explain_message_execvp(char *message, int message_size, const char *pathname, char *const *argv);
void explain_message_errno_execvp(char *message, int message_size, int errnum, const char *pathname, char *const *argv);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *execvp*(3) system call.

### explain_execvp

const char *explain_execvp(const char *pathname, char *const *argv);

The **explain_execvp** function is used to obtain an explanation of an error returned by the *execvp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (execvp(pathname, argv) < 0)
{
    fprintf(stderr, "%s\n", explain_execvp(pathname, argv));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_execvp_or_die*(3) function.

*pathname*
: The original pathname, exactly as passed to the *execvp*(3) system call.

*argv*
: The original argv, exactly as passed to the *execvp*(3) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_execvp

const char *explain_errno_execvp(int errnum, const char *pathname, char *const *argv);

The **explain_errno_execvp** function is used to obtain an explanation of an error returned by the *execvp*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (execvp(pathname, argv) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_execvp(err,
        pathname, argv));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_execvp_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *execvp*(3) system call.

*argv* The original argv, exactly as passed to the *execvp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_execvp**

void explain_message_execvp(char *message, int message_size, const char *pathname, char *const *argv);

The **explain_message_execvp** function may be used to obtain an explanation of an error returned by the *execvp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (execvp(pathname, argv) < 0)
{
    char message[3000];
    explain_message_execvp(message, sizeof(message), pathname, argv);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_execvp_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *execvp*(3) system call.

*argv* The original argv, exactly as passed to the *execvp*(3) system call.

**explain_message_errno_execvp**

void explain_message_errno_execvp(char *message, int message_size, int errnum, const char *pathname, char *const *argv);

The **explain_message_errno_execvp** function may be used to obtain an explanation of an error returned by the *execvp*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (execvp(pathname, argv) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_execvp(message, sizeof(message),
        err, pathname, argv);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
```

```
}
```

The above code example is available pre-packaged as the *explain_execvp_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *execvp*(3) system call.

*argv* The original argv, exactly as passed to the *execvp*(3) system call.

## SEE ALSO

*execvp*(3)
execute a file

*explain_execvp_or_die*(3)
execute a file and report errors

## COPYRIGHT

## NAME

explain_execvp_or_die − execute a file and report errors

## SYNOPSIS

#include <libexplain/execvp.h>

void explain_execvp_or_die(const char *pathname, char *const *argv);

## DESCRIPTION

The **explain_execvp_or_die** function is used to call the *execvp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_execvp*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_execvp_or_die(pathname, argv);
```

*pathname*
: The pathname, exactly as to be passed to the *execvp*(3) system call.

*argv*
: The argv, exactly as to be passed to the *execvp*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*execvp*(3)
: execute a file

*explain_execvp*(3)
: explain *execvp*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_exit − print an explanation of exit status before exiting

## SYNOPSIS

```
#include <libexplain/libexplain.h>

void explain_exit_on_exit(void);
void explain_exit_on_error(void);
void explain_exit_cancel(void);
```

## DESCRIPTION

The *explain_exit_on_exit* function may be used to have the calling program print an explanation of its exit status (the value passed to *exit*(3) or the return value from *main*) immediately before it terminates.

The *explain_exit_on_error* function may be used to have the calling program print an explanation of its exit status immediately before it terminates, if that exit status is not EXIT_SUCCESS.

The *explain_exit_cancel* function may be used to cancel the effect of the *explain_exit_on_exit* or *explain_exit_on_error* function.

These functions may be called multiple times, and in any order. The last called has precedence. The explanation will never be printed more than once.

### Call Exit As Normal

In order to have the explanation printed, simply call *exit*(3) as normal, or return from *main* as normal. Do not call any of these functions in order to exit your program, they are called before you exit your program.

### Caveat

This functionality is only available on systems with the *on_exit*(3) system call. Unfortunately, the *atexit*(3) system call is not sufficiently capable, as it does not pass the exit status to the registered function.

## SEE ALSO

*exit*(3) cause normal process termination

*atexit*(3) register a function to be called at normal process termination

*on_exit*(3)
register a function to be called at normal process termination

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fchdir − explain fchdir(2) errors

## SYNOPSIS

```
#include <libexplain/fchdir.h>
const char *explain_fchdir(int fildes);
void explain_message_fchdir(char *message, int message_size, int fildes);
const char *explain_errno_fchdir(int errnum, int fildes);
void explain_message_errno_fchdir(char *message, int message_size, int errnum, int fildes);
```

## DESCRIPTION

These functions may be used to obtain explanations for *fchdir*(2) errors.

### explain_fchdir

const char *explain_fchdir(int fildes);

The explain_fchdir function is used to obtain an explanation of an error returned by the *fchdir*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fchdir(fildes) < 0)
{
    fprintf(stderr, '%s0, explain_fchdir(fildes));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *fchdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fchdir

const char *explain_errno_fchdir(int errnum, int fildes);

The explain_errno_fchdir function is used to obtain an explanation of an error returned by the *fchdir*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fchdir(fildes) < 0)
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_fchdir(err, fildes));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fchdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fchdir**

void explain_message_fchdir(char *message, int message_size, int fildes);

The explain_message_fchdir function is used to obtain an explanation of an error returned by the *fchdir*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fchdir(fildes) < 0)
{
    char message[3000];
    explain_message_fchdir(message, sizeof(message), fildes);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fchdir*(2) system call.

**explain_message_errno_fchdir**

void explain_message_errno_fchdir(char *message, int message_size, int errnum, int fildes);

The explain_message_errno_fchdir function is used to obtain an explanation of an error returned by the *fchdir*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fchdir(fildes) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fchdir(message, sizeof(message), err,
        fildes);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fchdir*(2) system call.

## COPYRIGHT

## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fchdir_or_die − change directory and report errors

## SYNOPSIS

#include <libexplian/fchdir.h>
void explain_fchdir_or_die(int fildes);

## DESCRIPTION

The explain_fchdir_or_die function is used to change directory via the *fchdir*(2) system call. On failure, it prints an error message on stderr via *explain_fchdir*(3), and exits.

This function is intended to be used in a fashion similar to the following example:

```
explain_fchdir_or_die(fildes);
```

*fildes* exactly as to be passed to the *fchdir*(2) system call.

## SEE ALSO

*fchdir*(3)
change working directory

*explain_fchdir*(3)
report *fchdir*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fchmod − explain fchmod(2) errors

## SYNOPSIS

#include <libexplain/fchmod.h>
const char *explain_fchmod(int fildes, int mode);
const char *explain_errno_fchmod(int errnum, int fildes, int mode);
void explain_message_fchmod(char *message, int message_size, int fildes, int mode);
void explain_message_errno_fchmod(char *message, int message_size, int errnum, int fildes, int mode);

## DESCRIPTION

The explain_fchmod function may be used to obtain explanations for *fchmod*(2) errors.

### explain_fchmod

const char *explain_fchmod(int fildes, int mode);

The explain_fchmod function is used to obtain an explanation of an error returned by the *fchmod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fchmod(fildes, mode) < 0)
{
    fprintf(stderr, "%s\n", explain_fchmod(fildes, mode));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *fchmod*(2) system call.

*mode* The original mode, exactly as passed to the *fchmod*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fchmod

const char *explain_errno_fchmod(int errnum, int fildes, int mode);

The explain_errno_fchmod function is used to obtain an explanation of an error returned by the *fchmod*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fchmod(fildes, mode) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fchmod(err, fildes,
        mode));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fchmod*(2) system call.

*mode* The original mode, exactly as passed to the *fchmod*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fchmod**

void explain_message_fchmod(char *message, int message_size, int fildes, int mode);

The explain_message_fchmod function is used to obtain an explanation of an error returned by the *fchmod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fchmod(fildes, mode) < 0)
{
    char message[3000];
    explain_message_fchmod(message, sizeof(message), fildes, mode);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fchmod*(2) system call.

*mode* The original mode, exactly as passed to the *fchmod*(2) system call.

**explain_message_errno_fchmod**

void explain_message_errno_fchmod(char *message, int message_size, int errnum, int fildes, int mode);

The explain_message_errno_fchmod function is used to obtain an explanation of an error returned by the *fchmod*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fchmod(fildes, mode) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fchmod(message, sizeof(message), err,
        fildes, mode);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fchmod*(2) system call.

*mode* The original mode, exactly as passed to the *fchmod*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fchmod_or_die − change permissions of a file and report errors

## SYNOPSIS

#include <libexplain/libexplain.h>
void explain_fchmod_or_die(int fildes, int mode);

## DESCRIPTION

The **explain_fchmod_or_die** function is used to call the *fchmod*(2) system call. On failure an explanation wiil be printed to *stderr*, obtained from *explain_fchmod*(3), and the the process terminates by call ing `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_fchmod_or_die(fildes, mode);
```

*fildes* The fildes, exactly as to be passed to the *fchmod*(2) system call.

*mode* The mode, exactly as to be passed to the *fchmod*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exit(EXIT_FAILURE)s.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fchown − explain fchown(2) errors

## SYNOPSIS

#include <libexplain/fchown.h>

const char *explain_fchown(int fildes, int owner, int group);
const char *explain_errno_fchown(int errnum, int fildes, int owner, int group);
void explain_message_fchown(char *message, int message_size, int fildes, int owner, int group);
void explain_message_errno_fchown(char *message, int message_size, int errnum, int fildes, int owner, int group);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fchown*(2) system call.

### explain_fchown

const char *explain_fchown(int fildes, int owner, int group);

The **explain_fchown** function is used to obtain an explanation of an error returned by the *fchown*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fchown(fildes, owner, group) < 0)
{
    fprintf(stderr, "%s\n", explain_fchown(fildes, owner, group));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fchown_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *fchown*(2) system call.

*owner* The original owner, exactly as passed to the *fchown*(2) system call.

*group* The original group, exactly as passed to the *fchown*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fchown

const char *explain_errno_fchown(int errnum, int fildes, int owner, int group);

The **explain_errno_fchown** function is used to obtain an explanation of an error returned by the *fchown*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fchown(fildes, owner, group) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n",
        explain_errno_fchown(err, fildes, owner, group));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fchown_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fchown*(2) system call.

*owner* The original owner, exactly as passed to the *fchown*(2) system call.

*group* The original group, exactly as passed to the *fchown*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fchown**

void explain_message_fchown(char *message, int message_size, int fildes, int owner, int group);

The **explain_message_fchown** function may be used to obtain an explanation of an error returned by the *fchown*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fchown(fildes, owner, group) < 0)
{
    char message[3000];
    explain_message_fchown(message, sizeof(message), fildes, owner, group);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fchown_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fchown*(2) system call.

*owner* The original owner, exactly as passed to the *fchown*(2) system call.

*group* The original group, exactly as passed to the *fchown*(2) system call.

**explain_message_errno_fchown**

void explain_message_errno_fchown(char *message, int message_size, int errnum, int fildes, int owner, int group);

The **explain_message_errno_fchown** function may be used to obtain an explanation of an error returned by the *fchown*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fchown(fildes, owner, group) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fchown(message, sizeof(message),
        err, fildes, owner, group);
    fprintf(stderr, "%s\n", message);
```

```
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_fchown_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fchown*(2) system call.

*owner* The original owner, exactly as passed to the *fchown*(2) system call.

*group* The original group, exactly as passed to the *fchown*(2) system call.

## SEE ALSO

*fchown*(2)
change ownership of a file

*explain_fchown_or_die*(3)
change ownership of a file and report errors

## COPYRIGHT

## NAME

explain_fchown_or_die − change ownership of a file and report errors

## SYNOPSIS

#include <libexplain/fchown.h>

void explain_fchown_or_die(int fildes, int owner, int group);

## DESCRIPTION

The **explain_fchown_or_die** function is used to call the *fchown*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fchown*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_fchown_or_die(fildes, owner, group);
```

*fildes* The fildes, exactly as to be passed to the *fchown*(2) system call.

*owner* The owner, exactly as to be passed to the *fchown*(2) system call.

*group* The group, exactly as to be passed to the *fchown*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*fchown*(2)
change ownership of a file

*explain_fchown*(3)
explain *fchown*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fclose − explain fclose(3) errors

## SYNOPSIS

```
#include <libexplain/fclose.h>
const char *explain_fclose(FILE *fp);
const char *explain_errno_fclose(int errnum, FILE *fp);
void explain_message_fclose(char *message, int message_size, FILE *fp);
void explain_message_errno_fclose(char *message, int message_size, int errnum, FILE *fp);
```

## DESCRIPTION

These functions may be used to obtain explanations of *fclose*(3) errors.

### explain_fclose

const char *explain_fclose(FILE * fp);

The explain_fclose function is used to obtain an explanation of an error returned by the *fclose*(3) function. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fclose(fp))
{
    fprintf(stderr, "%s\n", explain_fclose(fp));
    exit(EXIT_FAILURE);
}
```

*fp* The original fp, exactly as passed to the *fclose*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Note:** This function may be of little diagnostic value, because libc may have destroyed any useful context, leaving nothing for libexplain to work with (this is true of glibc in particular). For files that are open for writing, you will obtain more useful information by first calling *fflush*(3), as in the following example

```
if (fflush(fp))
{
    fprintf(stderr, "%s\n", explain_fflush(fp));
    exit(EXIT_FAILURE);
}
if (fclose(fp))
{
    fprintf(stderr, "%s\n", explain_fclose(fp));
    exit(EXIT_FAILURE);
}
```

### explain_errno_fclose

const char *explain_errno_fclose(int errnum, FILE * fp);

The explain_errno_fclose function is used to obtain an explanation of an error returned by the *fclose*(3) function. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fclose(fp))
```

```
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fclose(err, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fclose*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Note:** This function may be of little diagnostic value, because libc may have destroyed any useful context, leaving nothing for libexplain to work with (this is true of glibc in particular). For files that are open for writing, you will obtain more useful information by first calling *fflush*(3), as in the following example

```
if (fflush(fp))
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fflush(err, fp));
    exit(EXIT_FAILURE);
}
if (fclose(fp))
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fclose(err, fp));
    exit(EXIT_FAILURE);
}
```

**explain_message_fclose**

void explain_message_fclose(char *message, int message_size, FILE *fp);

The explain_message_fclose function is used to obtain an explanation of an error returned by the *fclose*(3) function. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fclose(fp))
{
    char message[3000];
    explain_message_fclose(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fclose(3)* system call.

**Note:** This function may be of little diagnostic value, because libc may have destroyed any useful context,

leaving nothing for libexplain to work with (this is true of glibc in particular). For files that are open for writing, you will obtain more useful information by first calling *fflush*(3), as in the following example

```
if (fflush(fp))
{
    char message[3000];
    explain_message_fflush(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
if (fclose(fp))
{
    char message[3000];
    explain_message_fclose(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

**explain_message_errno_fclose**

void explain_message_errno_fclose(char *message, int message_size, int errnum, FILE *fp);

The explain_message_errno_fclose function is used to obtain an explanation of an error returned by the *fclose*(3) function. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following exameple:

```
if (fclose(fp))
{
    int err = errno;
    char message[3000];
    explain_message_errno_fclose(message, sizeof(message),
        err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fclose*(3) system call.

**Note:** This function may be of little diagnostic value, because libc may have destroyed any useful context, leaving nothing for libexplain to work with (this is true of glibc in particular). For files that are open for writing, you will obtain more useful information by first calling *fflush*(3), as in the following example

```
if (fflush(fp))
{
    int err = errno;
    char message[3000];
    explain_message_errno_fflush(message, sizeof(message),
        err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

```
if (fclose(fp))
{
    int err = errno;
    char message[3000];
    explain_message_errno_fclose(message, sizeof(message),
        err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fclose_or_die − close a stream and report errors

## SYNOPSIS

#include <libexplain/fclose.h>
void explain_fclose_or_die(FILE *fp);

## DESCRIPTION

The explain_fclose_or_die function is used to *fflush*(3) and *fclose*(3) the given stream. If there is an error, it will be reported using *explain_fclose*(3), and then terminates by calling `exit(EXIT_FAILURE)`.

```
explain_fclose_or_die(fp);
```

*fp* The fp, exactly as to be passed to the *fclose*(3) system call.

Returns: Only returns on success. Reports error and process exits on failure.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fcntl − explain fcntl(2) errors

## SYNOPSIS

#include <libexplain/fcntl.h>

const char *explain_fcntl(int fildes, int command, long arg);
const char *explain_errno_fcntl(int errnum, int fildes, int command, long arg);
void explain_message_fcntl(char *message, int message_size, int fildes, int command, long arg);
void explain_message_errno_fcntl(char *message, int message_size, int errnum, int fildes, int command, long arg);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fcntl*(2) system call.

### explain_fcntl

const char *explain_fcntl(int fildes, int command, long arg);

The **explain_fcntl** function is used to obtain an explanation of an error returned by the *fcntl*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fcntl(fildes, command, arg) < 0)
{
    fprintf(stderr, "%s\n", explain_fcntl(fildes, command, arg));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *fcntl*(2) system call.

*command*
The original command, exactly as passed to the *fcntl*(2) system call.

*arg* The original arg, exactly as passed to the *fcntl*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fcntl

const char *explain_errno_fcntl(int errnum, int fildes, int command, long arg);

The **explain_errno_fcntl** function is used to obtain an explanation of an error returned by the *fcntl*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fcntl(fildes, command, arg) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fcntl(err, fildes, command, arg));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fcntl*(2) system call.

*command*
The original command, exactly as passed to the *fcntl*(2) system call.

*arg* The original arg, exactly as passed to the *fcntl*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fcntl**

void explain_message_fcntl(char *message, int message_size, int fildes, int command, long arg);

The **explain_message_fcntl** function may be used to obtain an explanation of an error returned by the *fcntl*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fcntl(fildes, command, arg) < 0)
{
    char message[3000];
    explain_message_fcntl(message, sizeof(message), fildes, command, arg);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fcntl*(2) system call.

*command*
The original command, exactly as passed to the *fcntl*(2) system call.

*arg* The original arg, exactly as passed to the *fcntl*(2) system call.

**explain_message_errno_fcntl**

void explain_message_errno_fcntl(char *message, int message_size, int errnum, int fildes, int command, long arg);

The **explain_message_errno_fcntl** function may be used to obtain an explanation of an error returned by the *fcntl*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fcntl(fildes, command, arg) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fcntl(message, sizeof(message), err, fildes,
        command, arg);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fcntl*(2) system call.

*command*
The original command, exactly as passed to the *fcntl*(2) system call.

*arg* The original arg, exactly as passed to the *fcntl*(2) system call.

## SEE ALSO

*fcntl*(2) manipulate a file descriptor

*explain_fcntl_or_die*(3)
manipulate a file descriptor and report errors

## COPYRIGHT

## NAME

explain_fcntl_or_die − manipulate a file descriptor and report errors

## SYNOPSIS

#include <libexplain/fcntl.h>

int explain_fcntl_or_die(int fildes, int command, long arg);

## DESCRIPTION

The **explain_fcntl_or_die** function is used to call the *fcntl*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fcntl*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int result = explain_fcntl_or_die(fildes, command, arg);
```

*fildes* The fildes, exactly as to be passed to the *fcntl*(2) system call.

*command*
The command, exactly as to be passed to the *fcntl*(2) system call.

*arg* The arg, exactly as to be passed to the *fcntl*(2) system call.

Returns: This function only returns on success, and it returns whatever was returned by the fcntl(2) call; depending on the command, this may have no use. On failure, prints an explanation and exits, it does not return.

## SEE ALSO

*fcntl*(2) manipulate a file descriptor

*explain_fcntl*(3)
explain *fcntl*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fdopen − explain fdopen(3) errors

## SYNOPSIS

#include <libexplain/fdopen.h>

const char *explain_fdopen(int fildes, const char *flags);
const char *explain_errno_fdopen(int errnum, int fildes, const char *flags);
void explain_message_fdopen(char *message, int message_size, int fildes, const char *flags);
void explain_message_errno_fdopen(char *message, int message_size, int errnum, int fildes, const char *flags);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fdopen*(3) system call.

### explain_fdopen

const char *explain_fdopen(int fildes, const char *flags);

The **explain_fdopen** function is used to obtain an explanation of an error returned by the *fdopen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = fdopen(fildes, flags);
if (!fp)
{
    fprintf(stderr, "%s\n", explain_fdopen(fildes, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fdopen_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *fdopen*(3) system call.

*flags* The original flags, exactly as passed to the *fdopen*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fdopen

const char *explain_errno_fdopen(int errnum, int fildes, const char *flags);

The **explain_errno_fdopen** function is used to obtain an explanation of an error returned by the *fdopen*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = fdopen(fildes, flags);
if (!fp)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fdopen(err, fildes, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fdopen_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fdopen*(3) system call.

*flags* The original flags, exactly as passed to the *fdopen*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fdopen**

void explain_message_fdopen(char *message, int message_size, int fildes, const char *flags);

The **explain_message_fdopen** function may be used to obtain an explanation of an error returned by the *fdopen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = fdopen(fildes, flags);
if (!fp)
{
    char message[3000];
    explain_message_fdopen(message, sizeof(message), fildes, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fdopen_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fdopen*(3) system call.

*flags* The original flags, exactly as passed to the *fdopen*(3) system call.

**explain_message_errno_fdopen**

void explain_message_errno_fdopen(char *message, int message_size, int errnum, int fildes, const char *flags);

The **explain_message_errno_fdopen** function may be used to obtain an explanation of an error returned by the *fdopen*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = fdopen(fildes, flags);
if (!fp)

{
    int err = errno;
    char message[3000];
    explain_message_errno_fdopen(message, sizeof(message),
        err, fildes, flags);
    fprintf(stderr, "%s\n", message);
```

```
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_fdopen_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fdopen*(3) system call.

*flags* The original flags, exactly as passed to the *fdopen*(3) system call.

## SEE ALSO

*fdopen*(3)
stream open functions

*explain_fdopen_or_die*(3)
stream open functions and report errors

## COPYRIGHT

## NAME

explain_fdopendir − explain *fdopendir*(3) errors

## SYNOPSIS

#include <libexplain/fdopendir.h>

const char *explain_fdopendir(int fildes);
const char *explain_errno_fdopendir(int errnum, int fildes);
void explain_message_fdopendir(char *message, int message_size, int fildes);
void explain_message_errno_fdopendir(char *message, int message_size, int errnum, int fildes);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fdopendir*(3) system call.

### explain_fdopendir

const char *explain_fdopendir(int fildes);

The **explain_fdopendir** function is used to obtain an explanation of an error returned by the *fdopendir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *fdopendir*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
DIR *result = fdopendir(fildes);
if (!result)
{
   fprintf(stderr, "%s\n", explain_fdopendir(fildes));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fdopendir_or_die*(3) function.

### explain_errno_fdopendir

const char *explain_errno_fdopendir(int errnum, int fildes);

The **explain_errno_fdopendir** function is used to obtain an explanation of an error returned by the *fdopendir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fdopendir*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
DIR *result = fdopendir(fildes);
```

```
if (!result)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_fdopendir(err, fildes));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fdopendir_or_die*(3) function.

**explain_message_fdopendir**

void explain_message_fdopendir(char *message, int message_size, int fildes);

The **explain_message_fdopendir** function is used to obtain an explanation of an error returned by the *fdopendir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fdopendir*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
DIR *result = fdopendir(fildes);
if (!result)
{
    char message[3000];
   explain_message_fdopendir(message, sizeof(message), fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fdopendir_or_die*(3) function.

**explain_message_errno_fdopendir**

void explain_message_errno_fdopendir(char *message, int message_size, int errnum, int fildes);

The **explain_message_errno_fdopendir** function is used to obtain an explanation of an error returned by the *fdopendir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fdopendir*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
DIR *result = fdopendir(fildes);
if (!result)
{
    int err = errno;
    char message[3000];
   explain_message_errno_fdopendir(message, sizeof(message), err,
```

```
                fildes);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_fdopendir_or_die*(3) function.

## SEE ALSO

*fdopendir*(3)
open a directory

*explain_fdopendir_or_die*(3)
open a directory and report errors

## COPYRIGHT

## NAME

explain_fdopendir_or_die − open a directory and report errors

## SYNOPSIS

#include <libexplain/fdopendir.h>

DIR *explain_fdopendir_or_die(int fildes);
DIR *explain_fdopendir_on_error(int fildes);

## DESCRIPTION

The **explain_fdopendir_or_die** function is used to call the *fdopendir*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fdopendir*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fdopendir_on_error** function is used to call the *fdopendir*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fdopendir*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *fdopendir*(3) system call.

## RETURN VALUE

The **explain_fdopendir_or_die** function only returns on success, see *fdopendir*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fdopendir_on_error** function always returns the value return by the wrapped *fdopendir*(3) system call.

## EXAMPLE

The **explain_fdopendir_or_die** function is intended to be used in a fashion similar to the following example:

```
DIR *result = explain_fdopendir_or_die(fildes);
```

## SEE ALSO

*fdopendir*(3)
open a directory

*explain_fdopendir*(3)
explain *fdopendir*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fdopen_or_die − stream open functions and report errors

## SYNOPSIS

#include <libexplain/fdopen.h>

void explain_fdopen_or_die(int fd, const char *mode);

## DESCRIPTION

The **explain_fdopen_or_die** function is used to call the *fdopen*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fdopen*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = explain_fdopen_or_die(fd, mode);
```

*fd* The fd, exactly as to be passed to the *fdopen*(3) system call.

*mode* The mode, exactly as to be passed to the *fdopen*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*fdopen*(3)
stream open functions

*explain_fdopen*(3)
explain *fdopen*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_feof − explain *feof*(3) errors

## SYNOPSIS

#include <libexplain/feof.h>

const char *explain_feof(FILE *fp);
const char *explain_errno_feof(int errnum, FILE *fp);
void explain_message_feof(char *message, int message_size, FILE *fp);
void explain_message_errno_feof(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *feof*(3) system call.

### explain_feof

const char *explain_feof(FILE *fp);

The **explain_feof** function is used to obtain an explanation of an error returned by the *feof*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *feof*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (feof(fp) < 0)
{
   fprintf(stderr, "%s\n", explain_feof(fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_feof_or_die*(3) function.

### explain_errno_feof

const char *explain_errno_feof(int errnum, FILE *fp);

The **explain_errno_feof** function is used to obtain an explanation of an error returned by the *feof*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *feof*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (feof(fp) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_feof(err, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_feof_or_die*(3) function.

**explain_message_feof**

void explain_message_feof(char *message, int message_size, FILE *fp);

The **explain_message_feof** function is used to obtain an explanation of an error returned by the *feof*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *feof*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (feof(fp) < 0)
{
    char message[3000];
    explain_message_feof(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_feof_or_die*(3) function.

**explain_message_errno_feof**

void explain_message_errno_feof(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_feof** function is used to obtain an explanation of an error returned by the *feof*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *feof*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (feof(fp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_feof(message, sizeof(message), err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_feof_or_die*(3) function.

## SEE ALSO

*feof*(3) check and reset stream status

*explain_feof_or_die*(3)
check and reset stream status and report errors

## COPYRIGHT

## NAME

explain_feof_or_die − check and reset stream status and report errors

## SYNOPSIS

#include <libexplain/feof.h>

void explain_feof_or_die(FILE *fp);
int explain_feof_on_error(FILE *fp);

## DESCRIPTION

The **explain_feof_or_die** function is used to call the *feof*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_feof*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_feof_on_error** function is used to call the *feof*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_feof*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *feof*(3) system call.

## RETURN VALUE

The **explain_feof_or_die** function only returns on success, see *feof*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_feof_on_error** function always returns the value return by the wrapped *feof*(3) system call.

## EXAMPLE

The **explain_feof_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_feof_or_die(fp);
```

## SEE ALSO

*feof*(3) check and reset stream status

*explain_feof*(3)
explain *feof*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_ferror − explain ferror(3) errors

## SYNOPSIS

#include <libexplain/ferror.h>

const char *explain_ferror(FILE *fp);
const char *explain_errno_ferror(int errnum, FILE *fp);
void explain_message_ferror(char *message, int message_size, FILE *fp);
void explain_message_errno_ferror(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *ferror*(3) system call.

### explain_ferror

const char *explain_ferror(FILE *fp);

The **explain_ferror** function is used to obtain an explanation of an error returned by the *ferror*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (ferror(fp) < 0)
{
    fprintf(stderr, "%s\n", explain_ferror(fp));
    exit(EXIT_FAILURE);
}
```

It is essential that this function cal be placed as close as possible to the I/O code that has caused the problem, otherwise intervening code could have altered the *errno* global variable.

*fp* The original fp, exactly as passed to the *ferror*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_ferror

const char *explain_errno_ferror(int errnum, FILE *fp);

The **explain_errno_ferror** function is used to obtain an explanation of an error returned by the *ferror*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (ferror(fp) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_ferror(err, fp));
    exit(EXIT_FAILURE);
}
```

It is essential that this function cal be placed as close as possible to the I/O code that has caused the problem, otherwise intervening code could have altered the *errno* global variable.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *ferror*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_ferror**

void explain_message_ferror(char *message, int message_size, FILE *fp);

The **explain_message_ferror** function may be used to obtain an explanation of an error returned by the *ferror*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (ferror(fp) < 0)
{
    char message[3000];
    explain_message_ferror(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

It is essential that this function cal be placed as close as possible to the I/O code that has caused the problem, otherwise intervening code could have altered the *errno* global variable.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *ferror*(3) system call.

**explain_message_errno_ferror**

void explain_message_errno_ferror(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_ferror** function may be used to obtain an explanation of an error returned by the *ferror*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (ferror(fp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_ferror(message, sizeof(message), err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

It is essential that this function cal be placed as close as possible to the I/O code that has caused the problem, otherwise intervening code could have altered the *errno* global variable.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *ferror*(3) system call.

## SEE ALSO

*ferror*(3)
check stream status

*explain_ferror_or_die*(3)
check stream status and report errors

## COPYRIGHT

## NAME

explain_ferror_or_die − check stream status and report errors

## SYNOPSIS

#include <libexplain/ferror.h>

void explain_ferror_or_die(FILE *fp);

## DESCRIPTION

The **explain_ferror_or_die** function is used to call the *ferror*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_ferror*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_ferror_or_die(fp);
```

It is essential that this function cal be placed as close as possible to the I/O code that has caused the problem, otherwise intervening code could have altered the *errno* global variable.

*fp* The fp, exactly as to be passed to the *ferror*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*ferror*(3)
check stream status

*explain_ferror*(3)
explain *ferror*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fflush − explain fflush(3) errors

## SYNOPSIS

#include <libexplain/fflush.h>

const char *explain_fflush(FILE *fp);
const char *explain_errno_fflush(int errnum, FILE *fp);
void explain_message_fflush(char *message, int message_size, FILE *fp);
void explain_message_errno_fflush(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fflush*(3) system call.

### explain_fflush

const char *explain_fflush(FILE *fp);

The **explain_fflush** function is used to obtain an explanation of an error returned by the *fflush*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *fflush*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fflush(fp) < 0)
{
   fprintf(stderr, "%s\n", explain_fflush(fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fflush_or_die*(3) function.

### explain_errno_fflush

const char *explain_errno_fflush(int errnum, FILE *fp);

The **explain_errno_fflush** function is used to obtain an explanation of an error returned by the *fflush*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fflush*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fflush(fp) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fflush(err, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fflush_or_die*(3) function.

**explain_message_fflush**

void explain_message_fflush(char *message, int message_size, FILE *fp);

The **explain_message_fflush** function is used to obtain an explanation of an error returned by the *fflush*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fflush*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fflush(fp) < 0)
{
    char message[3000];
    explain_message_fflush(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fflush_or_die*(3) function.

**explain_message_errno_fflush**

void explain_message_errno_fflush(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_fflush** function is used to obtain an explanation of an error returned by the *fflush*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fflush*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fflush(fp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fflush(message, sizeof(message), err,
    fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fflush_or_die*(3) function.

## SEE ALSO

*fflush*(3) flush a stream

*explain_fflush_or_die*(3)
flush a stream and report errors

## COPYRIGHT

## NAME

explain_fflush_or_die − flush a stream and report errors

## SYNOPSIS

#include <libexplain/fflush.h>

void explain_fflush_or_die(FILE *fp);
int explain_fflush_on_error(FILE *fp);

## DESCRIPTION

The **explain_fflush_or_die** function is used to call the *fflush*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fflush*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fflush_on_error** function is used to call the *fflush*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fflush*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *fflush*(3) system call.

## RETURN VALUE

The **explain_fflush_or_die** function only returns on success, see *fflush*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fflush_on_error** function always returns the value return by the wrapped *fflush*(3) system call.

## EXAMPLE

The **explain_fflush_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fflush_or_die(fp);
```

## SEE ALSO

*fflush*(3) flush a stream

*explain_fflush*(3)
explain *fflush*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fgetc − explain fgetc(3) errors

## SYNOPSIS

#include <libexplain/fgetc.h>

const char *explain_fgetc(FILE *fp);
const char *explain_errno_fgetc(int errnum, FILE *fp);
void explain_message_fgetc(char *message, int message_size, FILE *fp);
void explain_message_errno_fgetc(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fgetc*(3) system call.

### explain_fgetc

const char *explain_fgetc(FILE *fp);

The **explain_fgetc** function is used to obtain an explanation of an error returned by the *fgetc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int c = fgetc(fp);
if (c == EOF && ferror(fp))
{
    fprintf(stderr, "%s\n", explain_fgetc(fp));
    exit(EXIT_FAILURE);
}
```

*fp* The original fp, exactly as passed to the *fgetc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fgetc

const char *explain_errno_fgetc(int errnum, FILE *fp);

The **explain_errno_fgetc** function is used to obtain an explanation of an error returned by the *fgetc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int c = fgetc(fp);
if (c == EOF && ferror(fp))
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fgetc(err, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fgetc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fgetc**

void explain_message_fgetc(char *message, int message_size, FILE *fp);

The **explain_message_fgetc** function may be used to obtain an explanation of an error returned by the *fgetc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int c = fgetc(fp);
if (c == EOF && ferror(fp))
{
    char message[3000];
    explain_message_fgetc(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fgetc*(3) system call.

**explain_message_errno_fgetc**

void explain_message_errno_fgetc(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_fgetc** function may be used to obtain an explanation of an error returned by the *fgetc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int c = fgetc(fp);
if (c == EOF && ferror(fp))
{
    int err = errno;
    char message[3000];
    explain_message_errno_fgetc(message, sizeof(message), err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fgetc*(3) system call.

## SEE ALSO

*fgetc*(3) input of characters

*explain_fgetc_or_die*(3)
input of characters and report errors

## COPYRIGHT

## NAME

explain_fgetc_or_die − input of characters and report errors

## SYNOPSIS

#include <libexplain/fgetc.h>

int explain_fgetc_or_die(FILE *fp);

## DESCRIPTION

The **explain_fgetc_or_die** function is used to call the *fgetc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fgetc*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int c = explain_fgetc_or_die(fp);
```

*fp* The fp, exactly as to be passed to the *fgetc*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*fgetc*(3) input of characters

*explain_fgetc*(3)
explain *fgetc*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fgetpos − explain *fgetpos*(3) errors

## SYNOPSIS

#include <libexplain/fgetpos.h>

const char *explain_fgetpos(FILE *fp, fpos_t *pos);
const char *explain_errno_fgetpos(int errnum, FILE *fp, fpos_t *pos);
void explain_message_fgetpos(char *message, int message_size, FILE *fp, fpos_t *pos);
void explain_message_errno_fgetpos(char *message, int message_size, int errnum, FILE *fp, fpos_t *pos);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fgetpos*(3) system call.

### explain_fgetpos

const char *explain_fgetpos(FILE *fp, fpos_t *pos);

The **explain_fgetpos** function is used to obtain an explanation of an error returned by the *fgetpos*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *fgetpos*(3) system call.

*pos* The original pos, exactly as passed to the *fgetpos*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fgetpos(fp, pos) < 0)
{
   fprintf(stderr, "%s\n", explain_fgetpos(fp, pos));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fgetpos_or_die*(3) function.

### explain_errno_fgetpos

const char *explain_errno_fgetpos(int errnum, FILE *fp, fpos_t *pos);

The **explain_errno_fgetpos** function is used to obtain an explanation of an error returned by the *fgetpos*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fgetpos*(3) system call.

*pos* The original pos, exactly as passed to the *fgetpos*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fgetpos(fp, pos) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fgetpos(err, fp, pos));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fgetpos_or_die*(3) function.

**explain_message_fgetpos**

void explain_message_fgetpos(char *message, int message_size, FILE *fp, fpos_t *pos);

The **explain_message_fgetpos** function is used to obtain an explanation of an error returned by the *fgetpos*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fgetpos*(3) system call.

*pos* The original pos, exactly as passed to the *fgetpos*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fgetpos(fp, pos) < 0)
{
    char message[3000];
    explain_message_fgetpos(message, sizeof(message), fp, pos);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fgetpos_or_die*(3) function.

**explain_message_errno_fgetpos**

void explain_message_errno_fgetpos(char *message, int message_size, int errnum, FILE *fp, fpos_t *pos);

The **explain_message_errno_fgetpos** function is used to obtain an explanation of an error returned by the *fgetpos*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fgetpos*(3) system call.

*pos* The original pos, exactly as passed to the *fgetpos*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fgetpos(fp, pos) < 0)
{
    int err = errno;
    char message[3000];
```

```
        explain_message_errno_fgetpos(message, sizeof(message), err,
        fp, pos);
         fprintf(stderr, "%s\n", message);
         exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_fgetpos_or_die*(3) function.

## SEE ALSO

*fgetpos*(3)
reposition a stream

*explain_fgetpos_or_die*(3)
reposition a stream and report errors

## COPYRIGHT

## NAME

explain_fgetpos_or_die − reposition a stream and report errors

## SYNOPSIS

#include <libexplain/fgetpos.h>

void explain_fgetpos_or_die(FILE *fp, fpos_t *pos);
int explain_fgetpos_on_error(FILE *fp, fpos_t *pos);

## DESCRIPTION

The **explain_fgetpos_or_die** function is used to call the *fgetpos*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fgetpos*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fgetpos_on_error** function is used to call the *fgetpos*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fgetpos*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *fgetpos*(3) system call.

*pos* The pos, exactly as to be passed to the *fgetpos*(3) system call.

## RETURN VALUE

The **explain_fgetpos_or_die** function only returns on success, see *fgetpos*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fgetpos_on_error** function always returns the value return by the wrapped *fgetpos*(3) system call.

## EXAMPLE

The **explain_fgetpos_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fgetpos_or_die(fp, pos);
```

## SEE ALSO

*fgetpos*(3)
reposition a stream

*explain_fgetpos*(3)
explain *fgetpos*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fgets − explain fgets(3) errors

## SYNOPSIS

#include <libexplain/fgets.h>

const char *explain_fgets(char *data, int data_size, FILE *fp);
const char *explain_errno_fgets(int errnum, char *data, int data_size, FILE *fp);
void explain_message_fgets(char *message, int message_size, char *data, int data_size, FILE *fp);
void explain_message_errno_fgets(char *message, int message_size, int errnum, char *data, int data_size, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fgets*(3) system call.

### explain_fgets

const char *explain_fgets(char *data, int data_size, FILE *fp);

The **explain_fgets** function is used to obtain an explanation of an error returned by the *fgets*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fgets(data, data_size, fp) < 0)
{
    fprintf(stderr, "%s\n", explain_fgets(data, data_size, fp));
    exit(EXIT_FAILURE);
}
```

*data* The original data, exactly as passed to the *fgets*(3) system call.

*data_size*
The original data_size, exactly as passed to the *fgets*(3) system call.

*fp* The original fp, exactly as passed to the *fgets*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fgets

const char *explain_errno_fgets(int errnum, char *data, int data_size, FILE *fp);

The **explain_errno_fgets** function is used to obtain an explanation of an error returned by the *fgets*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fgets(data, data_size, fp) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fgets(err, data, data_size, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *fgets*(3) system call.

*data_size*
The original data_size, exactly as passed to the *fgets*(3) system call.

*fp* The original fp, exactly as passed to the *fgets*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fgets**

void explain_message_fgets(char *message, int message_size, char *data, int data_size, FILE *fp);

The **explain_message_fgets** function may be used to obtain an explanation of an error returned by the *fgets*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fgets(data, data_size, fp) < 0)
{
    char message[3000];
    explain_message_fgets(message, sizeof(message), data, data_size, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *fgets*(3) system call.

*data_size*
The original data_size, exactly as passed to the *fgets*(3) system call.

*fp* The original fp, exactly as passed to the *fgets*(3) system call.

**explain_message_errno_fgets**

void explain_message_errno_fgets(char *message, int message_size, int errnum, char *data, int data_size, FILE *fp);

The **explain_message_errno_fgets** function may be used to obtain an explanation of an error returned by the *fgets*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fgets(data, data_size, fp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fgets(message, sizeof(message), err,
        data, data_size, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *fgets*(3) system call.

*data_size*
The original data_size, exactly as passed to the *fgets*(3) system call.

*fp* The original fp, exactly as passed to the *fgets*(3) system call.

## SEE ALSO

*fgets*(3) input of strings

*explain_fgets_or_die*(3)
input of strings and report errors

## COPYRIGHT

## NAME

explain_fgets_or_die − input of strings and report errors

## SYNOPSIS

#include <libexplain/fgets.h>

char *explain_fgets_or_die(char *data, int data_size, FILE *fp);

## DESCRIPTION

The **explain_fgets_or_die** function is used to call the *fgets*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fgets*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_fgets_or_die(data, data_size, fp);
```

*data* The data, exactly as to be passed to the *fgets*(3) system call.

*data_size*
The data_size, exactly as to be passed to the *fgets*(3) system call.

*fp* The fp, exactly as to be passed to the *fgets*(3) system call.

Returns: This function only returns on success; data when a line is read, or NULL on end-of-file. On failure, prints an explanation and exits.

## SEE ALSO

*fgets*(3) input of strings

*explain_fgets*(3)
explain *fgets*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_filename_from_fildes − obtain filename from file descriptor

## SYNOPSIS

```
#include <libexplain/filename.h>

int explain_filename_from_fildes(int fildes, char *data, size_t data_size);

int explain_filename_from_stream(FILE *stream, char *data, size_t data_size);
```

## DESCRIPTION

The *explain_filename_from_fildes* function may be used to obtain the name of the file associated with the file descriptor.

The *explain_filename_from_stream* function may be used to obtain the name of the file associated with a file stream.

The filename is returned in the array pointed to by *data*. The filename will always be NUL terminated. If the returned filename is longer than *data_size*, it will be silently truncated; a size of at least (PATH_MAX + 1) is suggested.

On success, returns zero. If the file name cannot be determined, returns −1 (but does **not** set *errno*.)

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fileno − explain fileno(3) errors

## SYNOPSIS

#include <libexplain/fileno.h>

const char *explain_fileno(FILE *fp);
const char *explain_errno_fileno(int errnum, FILE *fp);
void explain_message_fileno(char *message, int message_size, FILE *fp);
void explain_message_errno_fileno(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fileno*(3) system call.

### explain_fileno

const char *explain_fileno(FILE *fp);

The **explain_fileno** function is used to obtain an explanation of an error returned by the *fileno*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *fileno*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fileno(fp) < 0)
{
   fprintf(stderr, "%s\n", explain_fileno(fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fileno_or_die*(3) function.

### explain_errno_fileno

const char *explain_errno_fileno(int errnum, FILE *fp);

The **explain_errno_fileno** function is used to obtain an explanation of an error returned by the *fileno*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fileno*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fileno(fp) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fileno(err, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fileno_or_die*(3) function.

**explain_message_fileno**

void explain_message_fileno(char *message, int message_size, FILE *fp);

The **explain_message_fileno** function is used to obtain an explanation of an error returned by the *fileno*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fileno*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fileno(fp) < 0)
{
    char message[3000];
    explain_message_fileno(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fileno_or_die*(3) function.

**explain_message_errno_fileno**

void explain_message_errno_fileno(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_fileno** function is used to obtain an explanation of an error returned by the *fileno*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fileno*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fileno(fp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fileno(message, sizeof(message), err,
    fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fileno_or_die*(3) function.

## SEE ALSO

*fileno*(3) check and reset stream status

*explain_fileno_or_die*(3)
check and reset stream status and report errors

## COPYRIGHT

## NAME

explain_fileno_or_die − check and reset stream status and report errors

## SYNOPSIS

#include <libexplain/fileno.h>

int explain_fileno_or_die(FILE *fp);
int explain_fileno_on_error(FILE *fp);

## DESCRIPTION

The **explain_fileno_or_die** function is used to call the *fileno*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fileno*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fileno_on_error** function is used to call the *fileno*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fileno*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *fileno*(3) system call.

## RETURN VALUE

The **explain_fileno_or_die** function only returns on success, see *fileno*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fileno_on_error** function always returns the value return by the wrapped *fileno*(3) system call.

## EXAMPLE

The **explain_fileno_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fileno_or_die(fp);
```

## SEE ALSO

*fileno*(3) check and reset stream status

*explain_fileno*(3)
explain *fileno*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_flock − explain flock(2) errors

## SYNOPSIS

#include <libexplain/flock.h>

const char *explain_flock(int fildes, int command);
const char *explain_errno_flock(int errnum, int fildes, int command);
void explain_message_flock(char *message, int message_size, int fildes, int command);
void explain_message_errno_flock(char *message, int message_size, int errnum, int fildes, int command);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *flock*(2) system call.

### explain_flock

const char *explain_flock(int fildes, int command);

The **explain_flock** function is used to obtain an explanation of an error returned by the *flock*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *flock*(2) system call.

*command*
The original command, exactly as passed to the *flock*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (flock(fildes, command) < 0)
{
   fprintf(stderr, "%s\n", explain_flock(fildes, command));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_flock_or_die*(3) function.

### explain_errno_flock

const char *explain_errno_flock(int errnum, int fildes, int command);

The **explain_errno_flock** function is used to obtain an explanation of an error returned by the *flock*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *flock*(2) system call.

*command*
The original command, exactly as passed to the *flock*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (flock(fildes, command) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_flock(err, fildes,
   command));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_flock_or_die*(3) function.

**explain_message_flock**

void explain_message_flock(char *message, int message_size, int fildes, int command);

The **explain_message_flock** function is used to obtain an explanation of an error returned by the *flock*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *flock*(2) system call.

*command*
The original command, exactly as passed to the *flock*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (flock(fildes, command) < 0)
{
    char message[3000];
   explain_message_flock(message, sizeof(message), fildes,
   command);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_flock_or_die*(3) function.

**explain_message_errno_flock**

void explain_message_errno_flock(char *message, int message_size, int errnum, int fildes, int command);

The **explain_message_errno_flock** function is used to obtain an explanation of an error returned by the *flock*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *flock*(2) system call.

*command*
The original command, exactly as passed to the *flock*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (flock(fildes, command) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_flock(message, sizeof(message), err,
   fildes, command);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_flock_or_die*(3) function.

## SEE ALSO

*flock*(2) apply or remove an advisory lock on an open file

*explain_flock_or_die*(3)
apply or remove an advisory lock on an open file and report errors

## COPYRIGHT

## NAME

explain_flock_or_die − control advisory lock on open file and report errors

## SYNOPSIS

#include <libexplain/flock.h>

void explain_flock_or_die(int fildes, int command);
int explain_flock_on_error(int fildes, int command))

## DESCRIPTION

The **explain_flock_or_die** function is used to call the *flock*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_flock*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_flock_on_error** function is used to call the *flock*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_flock*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *flock*(2) system call.

*command*
The command, exactly as to be passed to the *flock*(2) system call.

## RETURN VALUE

The **explain_flock_or_die** function only returns on success, see *flock*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_flock_on_error** function always returns the value return by the wrapped *flock*(2) system call.

## EXAMPLE

The **explain_flock_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_flock_or_die(fildes, command);
```

## SEE ALSO

*flock*(2) apply or remove an advisory lock on an open file

*explain_flock*(3)
explain *flock*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fopen − explain fopen(3) errors

## SYNOPSIS

#include <libexplain/fopen.h>
const char *explain_fopen(const char *path, const char *mode);
const char *explain_errno_fopen(int errnum, const char *path, const char *mode);
void explain_message_fopen(char *message, int message_size, const char *path, const char *mode);
void explain_message_errno_fopen(char *message, int message_size, int errnum, const char *path, const char *mode);

## DESCRIPTION

These functions may be used to obtain explanations for *fopen*(3) errors.

### explain_fopen

const char *explain_fopen(const char *path, const char *mode);

The explain_fopen function is used to obtain an explanation of an error returned by the *fopen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = fopen(path, mode);
if (!fp)
{
    const char *message = explain_fopen(path, mode);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*path* The original path, exactly as passed to the *fopen*(3) system call.

*mode* The original mode, exactly as passed to the *fopen*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fopen

const char *explain_errno_fopen(int errnum, const char *path, const char *mode);

The explain_errno_fopen function is used to obtain an explanation of an error returned by the *fopen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = fopen(path, mode);
if (!fp)
{
    const char *message = explain_errno_fopen(err, path, mode);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*path* The original path, exactly as passed to the *fopen*(3) system call.

*mode* The original mode, exactly as passed to the *fopen*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fopen**

void explain_message_fopen(char *message, int message_size, const char *path, const char *mode);

The explain_message_fopen function is used to obtain an explanation of an error returned by the *fopen*(3) system call. The least the message will contain is the value of strerror(errno), but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = fopen(path, mode);
if (!fp)
{
    char message[3000];
    explain_message_fopen(message, sizeof(message), path, mode);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*path* The original path, exactly as passed to the *fopen*(3) system call.

*mode* The original mode, exactly as passed to the *fopen*(3) system call

**explain_message_errno_fopen**

void explain_message_errno_fopen(char *message, int message_size, int errnum, const char *path, const char *mode);

The explain_message_errno_fopen function is used to obtain an explanation of an error returned by the *fopen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = fopen(path, mode);
if (!fp)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fopen(message, sizeof(message), err, path,
        mode);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*path*
: The original path, exactly as passed to the *fopen*(3) system call.

*mode*
: The original mode, exactly as passed to the *fopen*(3) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fopen_or_die − open file and report errors

## SYNOPSIS

```
#include <libexplain/fopen.h>

FILE *explain_fopen_or_die(const char *pathname, const char *flags);
```

## DESCRIPTION

The **explain_fopen_or_die**() function opens the file whose name is the string pointed to by pathname and associates a stream with it. See *fopen*(3) for more information.

This is a quick and simple way for programs to constitently report file open errors in a consistent and detailed fahion.

## RETURN VALUE

Upon successful completion **explain_fopen_or_die** returns a *FILE* pointer.

If an error occurs, **explain_fopen** will be called to explain the error, which will be printed onto *stderr*, and then the process will terminate by calling `exit(EXIT_FAILURE)`.

## SEE ALSO

*fopen*(3) stream open functions

*explain_fopen*(3)
explain *fopen*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fork − explain fork(2) errors

## SYNOPSIS

#include <libexplain/fork.h>

const char *explain_fork(void);
const char *explain_errno_fork(int errnum);
void explain_message_fork(char *message, int message_size);
void explain_message_errno_fork(char *message, int message_size, int errnum);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fork*(2) system call.

### explain_fork

const char *explain_fork(void);

The **explain_fork** function is used to obtain an explanation of an error returned by the *fork*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fork() < 0)
{
    fprintf(stderr, "%s\n", explain_fork());
    exit(EXIT_FAILURE);
}
```

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fork

const char *explain_errno_fork(int errnum);

The **explain_errno_fork** function is used to obtain an explanation of an error returned by the *fork*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fork() < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fork(err, ));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fork**

void explain_message_fork(char *message, int message_size);

The **explain_message_fork** function may be used to obtain an explanation of an error returned by the *fork*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fork() < 0)
{
    char message[3000];
    explain_message_fork(message, sizeof(message), );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

**explain_message_errno_fork**

void explain_message_errno_fork(char *message, int message_size, int errnum);

The **explain_message_errno_fork** function may be used to obtain an explanation of an error returned by the *fork*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fork() < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fork(message, sizeof(message), err, );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

## SEE ALSO

*fork*(2) create a child process

*explain_fork_or_die*(3)
create a child process and report errors

## COPYRIGHT

## NAME

explain_fork_or_die − create a child process and report errors

## SYNOPSIS

#include <libexplain/fork.h>

void explain_fork_or_die(void);

## DESCRIPTION

The **explain_fork_or_die** function is used to call the *fork*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fork*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_fork_or_die();
```

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*fork*(2) create a child process

*explain_fork*(3)
explain *fork*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fpathconf − explain fpathconf(3) errors

## SYNOPSIS

#include <libexplain/fpathconf.h>

const char *explain_fpathconf(int fildes, int name);
const char *explain_errno_fpathconf(int errnum, int fildes, int name);
void explain_message_fpathconf(char *message, int message_size, int fildes, int name);
void explain_message_errno_fpathconf(char *message, int message_size, int errnum, int fildes, int name);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fpathconf*(3) system call.

### explain_fpathconf

const char *explain_fpathconf(int fildes, int name);

The **explain_fpathconf** function is used to obtain an explanation of an error returned by the *fpathconf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fpathconf(fildes, name) < 0)
{
    fprintf(stderr, "%s\n", explain_fpathconf(fildes, name));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fpathconf_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *fpathconf*(3) system call.

*name* The original name, exactly as passed to the *fpathconf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fpathconf

const char *explain_errno_fpathconf(int errnum, int fildes, int name);

The **explain_errno_fpathconf** function is used to obtain an explanation of an error returned by the *fpathconf*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fpathconf(fildes, name) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fpathconf(err, fildes, name));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fpathconf_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fpathconf*(3) system call.

*name* The original name, exactly as passed to the *fpathconf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fpathconf**

void explain_message_fpathconf(char *message, int message_size, int fildes, int name);

The **explain_message_fpathconf** function may be used to obtain an explanation of an error returned by the *fpathconf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fpathconf(fildes, name) < 0)
{
    char message[3000];
    explain_message_fpathconf(message, sizeof(message), fildes, name);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fpathconf_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fpathconf*(3) system call.

*name* The original name, exactly as passed to the *fpathconf*(3) system call.

**explain_message_errno_fpathconf**

void explain_message_errno_fpathconf(char *message, int message_size, int errnum, int fildes, int name);

The **explain_message_errno_fpathconf** function may be used to obtain an explanation of an error returned by the *fpathconf*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fpathconf(fildes, name) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fpathconf(message, sizeof(message),
        err, fildes, name);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fpathconf_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes*
: The original fildes, exactly as passed to the *fpathconf*(3) system call.

*name*
: The original name, exactly as passed to the *fpathconf*(3) system call.

## SEE ALSO

*fpathconf*(3)
: get configuration values for files

*explain_fpathconf_or_die*(3)
: get configuration values for files and report errors

## COPYRIGHT

## NAME

explain_fpathconf_or_die − get file configuration and report errors

## SYNOPSIS

#include <libexplain/fpathconf.h>

long explain_fpathconf_or_die(int fildes, int name);

## DESCRIPTION

The **explain_fpathconf_or_die** function is used to call the *fpathconf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fpathconf*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
long result = explain_fpathconf_or_die(fildes, name);
```

*fildes* The fildes, exactly as to be passed to the *fpathconf*(3) system call.

*name* The name, exactly as to be passed to the *fpathconf*(3) system call.

Returns: This function only returns on success, see *fpathconf*(3) for more information. On failure, prints an explanation and exits.

## SEE ALSO

*fpathconf*(3)
get configuration values for files

*explain_fpathconf*(3)
explain *fpathconf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fprintf − explain *fprintf*(3) errors

## SYNOPSIS

#include <libexplain/fprintf.h>

const char *explain_fprintf(FILE *fp, const char *format, ...);
const char *explain_errno_fprintf(int errnum, FILE *fp, const char *format, ...);
void explain_message_fprintf(char *message, int message_size, FILE *fp, const char *format, ....);
void explain_message_errno_fprintf(char *message, int message_size, int errnum, FILE *fp, const char *format, ...);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fprintf*(3) system call.

### explain_fprintf

const char *explain_fprintf(FILE *fp, const char *format, ...);

The **explain_fprintf** function is used to obtain an explanation of an error returned by the *fprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *fprintf*(3) system call.

*format* The original format, exactly as passed to the *fprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = fprintf(fp, format, ...);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_fprintf(fp, format, ...));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fprintf_or_die*(3) function.

### explain_errno_fprintf

const char *explain_errno_fprintf(int errnum, FILE *fp, const char *format, ...);

The **explain_errno_fprintf** function is used to obtain an explanation of an error returned by the *fprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fprintf*(3) system call.

*format* The original format, exactly as passed to the *fprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = fprintf(fp, format, ...);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_fprintf(err, fp, format,
   ...));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fprintf_or_die*(3) function.

**explain_message_fprintf**

void explain_message_fprintf(char *message, int message_size, FILE *fp, const char *format, ...);

The **explain_message_fprintf** function is used to obtain an explanation of an error returned by the *fprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fprintf*(3) system call.

*format* The original format, exactly as passed to the *fprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = fprintf(fp, format, ...);
if (result < 0)
{
    char message[3000];
   explain_message_fprintf(message, sizeof(message), fp, format,
   ...);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fprintf_or_die*(3) function.

**explain_message_errno_fprintf**

void explain_message_errno_fprintf(char *message, int message_size, int errnum, FILE *fp, const char *format, ...);

The **explain_message_errno_fprintf** function is used to obtain an explanation of an error returned by the *fprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fprintf*(3) system call.

*format* The original format, exactly as passed to the *fprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = fprintf(fp, format, ...);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_fprintf(message, sizeof(message), err,
   fp, format, ...);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fprintf_or_die*(3) function.

## SEE ALSO

*fprintf*(3)
formatted output conversion

*explain_fprintf_or_die*(3)
formatted output conversion and report errors

## COPYRIGHT

## NAME

explain_fprintf_or_die − formatted output conversion and report errors

## SYNOPSIS

#include <libexplain/fprintf.h>

int explain_fprintf_or_die(FILE *fp, const char *format, ...);
int explain_fprintf_on_error(FILE *fp, const char *format, ...);

## DESCRIPTION

The **explain_fprintf_or_die** function is used to call the *fprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fprintf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fprintf_on_error** function is used to call the *fprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fprintf*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *fprintf*(3) system call.

*format* The format, exactly as to be passed to the *fprintf*(3) system call.

## RETURN VALUE

The **explain_fprintf_or_die** function only returns on success, see *fprintf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fprintf_on_error** function always returns the value return by the wrapped *fprintf*(3) system call.

## EXAMPLE

The **explain_fprintf_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_fprintf_or_die(fp, format, ...);
```

## SEE ALSO

*fprintf*(3)
formatted output conversion

*explain_fprintf*(3)
explain *fprintf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fpurge − explain *fpurge*(3) errors

## SYNOPSIS

#include <libexplain/fpurge.h>

const char *explain_fpurge(FILE *fp);
const char *explain_errno_fpurge(int errnum, FILE *fp);
void explain_message_fpurge(char *message, int message_size, FILE *fp);
void explain_message_errno_fpurge(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fpurge*(3) system call.

### explain_fpurge

const char *explain_fpurge(FILE *fp);

The **explain_fpurge** function is used to obtain an explanation of an error returned by the *fpurge*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *fpurge*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fpurge(fp) < 0)
{
   fprintf(stderr, "%s\n", explain_fpurge(fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fpurge_or_die*(3) function.

### explain_errno_fpurge

const char *explain_errno_fpurge(int errnum, FILE *fp);

The **explain_errno_fpurge** function is used to obtain an explanation of an error returned by the *fpurge*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fpurge*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fpurge(fp) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fpurge(err, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fpurge_or_die*(3) function.

**explain_message_fpurge**

void explain_message_fpurge(char *message, int message_size, FILE *fp);

The **explain_message_fpurge** function is used to obtain an explanation of an error returned by the *fpurge*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fpurge*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fpurge(fp) < 0)
{
    char message[3000];
    explain_message_fpurge(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fpurge_or_die*(3) function.

**explain_message_errno_fpurge**

void explain_message_errno_fpurge(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_fpurge** function is used to obtain an explanation of an error returned by the *fpurge*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fpurge*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fpurge(fp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fpurge(message, sizeof(message), err,
    fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fpurge_or_die*(3) function.

## SEE ALSO

*fpurge*(3)
purge a stream

*explain_fpurge_or_die*(3)
purge a stream and report errors

## COPYRIGHT

## NAME

explain_fpurge_or_die − purge a stream and report errors

## SYNOPSIS

#include <libexplain/fpurge.h>

void explain_fpurge_or_die(FILE *fp);
int explain_fpurge_on_error(FILE *fp);

## DESCRIPTION

The **explain_fpurge_or_die** function is used to call the *fpurge*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fpurge*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fpurge_on_error** function is used to call the *fpurge*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fpurge*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *fpurge*(3) system call.

## RETURN VALUE

The **explain_fpurge_or_die** function only returns on success, see *fpurge*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fpurge_on_error** function always returns the value return by the wrapped *fpurge*(3) system call.

## EXAMPLE

The **explain_fpurge_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fpurge_or_die(fp);
```

## SEE ALSO

*fpurge*(3)
purge a stream

*explain_fpurge*(3)
explain *fpurge*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fputc − explain fputc(3) errors

## SYNOPSIS

#include <libexplain/fputc.h>

const char *explain_fputc(int c, FILE *fp);
const char *explain_errno_fputc(int errnum, int c, FILE *fp);
void explain_message_fputc(char *message, int message_size, int c, FILE *fp);
void explain_message_errno_fputc(char *message, int message_size, int errnum, int c, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fputc*(3) system call.

### explain_fputc

const char *explain_fputc(int c, FILE *fp);

The **explain_fputc** function is used to obtain an explanation of an error returned by the *fputc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fputc(c, fp) == EOF)
{
    fprintf(stderr, "%s\n", explain_fputc(c, fp));
    exit(EXIT_FAILURE);
}
```

*c* The original c, exactly as passed to the *fputc*(3) system call.

*fp* The original fp, exactly as passed to the *fputc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fputc

const char *explain_errno_fputc(int errnum, int c, FILE *fp);

The **explain_errno_fputc** function is used to obtain an explanation of an error returned by the *fputc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fputc(c, fp) == EOF)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fputc(err, c, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*c* The original c, exactly as passed to the *fputc*(3) system call.

*fp* The original fp, exactly as passed to the *fputc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fputc**

void explain_message_fputc(char *message, int message_size, int c, FILE *fp);

The **explain_message_fputc** function may be used to obtain an explanation of an error returned by the *fputc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fputc(c, fp) == EOF)
{
    char message[3000];
    explain_message_fputc(message, sizeof(message), c, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*c* The original c, exactly as passed to the *fputc*(3) system call.

*fp* The original fp, exactly as passed to the *fputc*(3) system call.

**explain_message_errno_fputc**

void explain_message_errno_fputc(char *message, int message_size, int errnum, int c, FILE *fp);

The **explain_message_errno_fputc** function may be used to obtain an explanation of an error returned by the *fputc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fputc(c, fp) == EOF)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fputc(message, sizeof(message), err, c, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*c* The original c, exactly as passed to the *fputc*(3) system call.

*fp* The original fp, exactly as passed to the *fputc*(3) system call.

## SEE ALSO

*fputc*(3) output of characters

*explain_fputc_or_die*(3)
output of characters and report errors

## COPYRIGHT

## NAME

explain_fputc_or_die − output of characters and report errors

## SYNOPSIS

#include <libexplain/fputc.h>

void explain_fputc_or_die(int c, FILE *fp);

## DESCRIPTION

The **explain_fputc_or_die** function is used to call the *fputc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fputc*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_fputc_or_die(c, fp);
```

*c* The c, exactly as to be passed to the *fputc*(3) system call.

*fp* The fp, exactly as to be passed to the *fputc*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*fputc*(3) output of characters

*explain_fputc*(3)
explain *fputc*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fputs − explain *fputs*(3) errors

## SYNOPSIS

#include <libexplain/fputs.h>

const char *explain_fputs(const char *s, FILE *fp);
const char *explain_errno_fputs(int errnum, const char *s, FILE *fp);
void explain_message_fputs(char *message, int message_size, const char *s, FILE *fp);
void explain_message_errno_fputs(char *message, int message_size, int errnum, const char *s, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fputs*(3) system call.

### explain_fputs

const char *explain_fputs(const char *s, FILE *fp);

The **explain_fputs** function is used to obtain an explanation of an error returned by the *fputs*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*s* The original s, exactly as passed to the *fputs*(3) system call.

*fp* The original fp, exactly as passed to the *fputs*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fputs(s, fp) < 0)
{
   fprintf(stderr, "%s\n", explain_fputs(s, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fputs_or_die*(3) function.

### explain_errno_fputs

const char *explain_errno_fputs(int errnum, const char *s, FILE *fp);

The **explain_errno_fputs** function is used to obtain an explanation of an error returned by the *fputs*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*s* The original s, exactly as passed to the *fputs*(3) system call.

*fp* The original fp, exactly as passed to the *fputs*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fputs(s, fp) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_fputs(err, s, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fputs_or_die*(3) function.

**explain_message_fputs**

void explain_message_fputs(char *message, int message_size, const char *s, FILE *fp);

The **explain_message_fputs** function is used to obtain an explanation of an error returned by the *fputs*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*s* The original s, exactly as passed to the *fputs*(3) system call.

*fp* The original fp, exactly as passed to the *fputs*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fputs(s, fp) < 0)
{
    char message[3000];
   explain_message_fputs(message, sizeof(message), s, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fputs_or_die*(3) function.

**explain_message_errno_fputs**

void explain_message_errno_fputs(char *message, int message_size, int errnum, const char *s, FILE *fp);

The **explain_message_errno_fputs** function is used to obtain an explanation of an error returned by the *fputs*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*s* The original s, exactly as passed to the *fputs*(3) system call.

*fp* The original fp, exactly as passed to the *fputs*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fputs(s, fp) < 0)
{
    int err = errno;
    char message[3000];
```

```
        explain_message_errno_fputs(message, sizeof(message), err, s,
        fp);
         fprintf(stderr, "%s\n", message);
         exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_fputs_or_die*(3) function.

## SEE ALSO

*fputs*(3) write a string to a stream

*explain_fputs_or_die*(3)
write a string to a stream and report errors

## COPYRIGHT

## NAME

explain_fputs_or_die − write a string to a stream and report errors

## SYNOPSIS

#include <libexplain/fputs.h>

void explain_fputs_or_die(const char *s, FILE *fp);
int explain_fputs_on_error(const char *s, FILE *fp);

## DESCRIPTION

The **explain_fputs_or_die** function is used to call the *fputs*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fputs*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fputs_on_error** function is used to call the *fputs*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fputs*(3) function, but still returns to the caller.

*s* The s, exactly as to be passed to the *fputs*(3) system call.

*fp* The fp, exactly as to be passed to the *fputs*(3) system call.

## RETURN VALUE

The **explain_fputs_or_die** function only returns on success, see *fputs*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fputs_on_error** function always returns the value return by the wrapped *fputs*(3) system call.

## EXAMPLE

The **explain_fputs_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fputs_or_die(s, fp);
```

## SEE ALSO

*fputs*(3) write a string to a stream

*explain_fputs*(3)
explain *fputs*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fread − explain fread(3) errors

## SYNOPSIS

#include <libexplain/fread.h>

const char *explain_fread(void *ptr, size_t size, size_t nmemb, FILE *fp);
const char *explain_errno_fread(int errnum, void *ptr, size_t size, size_t nmemb, FILE *fp);
void explain_message_fread(char *message, int message_size, void *ptr, size_t size, size_t nmemb, FILE *fp);
void explain_message_errno_fread(char *message, int message_size, int errnum, void *ptr, size_t size, size_t nmemb, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fread*(3) system call.

### explain_fread

const char *explain_fread(void *ptr, size_t size, size_t nmemb, FILE *fp);

The **explain_fread** function is used to obtain an explanation of an error returned by the *fread*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
size_t how_many = fread(ptr, size, nmemb, fp);
if (how_many == 0 && ferror(fp))
{
    fprintf(stderr, "%s\n", explain_fread(ptr, size, nmemb, fp));
    exit(EXIT_FAILURE);
}
```

*ptr* The original ptr, exactly as passed to the *fread*(3) system call.

*size* The original size, exactly as passed to the *fread*(3) system call.

*nmemb* The original nmemb, exactly as passed to the *fread*(3) system call.

*fp* The original fp, exactly as passed to the *fread*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fread

const char *explain_errno_fread(int errnum, void *ptr, size_t size, size_t nmemb, FILE *fp);

The **explain_errno_fread** function is used to obtain an explanation of an error returned by the *fread*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
size_t how_many = fread(ptr, size, nmemb, fp);
if (how_many == 0 && ferror(fp))
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fread(err, ptr, size, nmemb, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ptr* The original ptr, exactly as passed to the *fread*(3) system call.

*size* The original size, exactly as passed to the *fread*(3) system call.

*nmemb* The original nmemb, exactly as passed to the *fread*(3) system call.

*fp* The original fp, exactly as passed to the *fread*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fread**

void explain_message_fread(char *message, int message_size, void *ptr, size_t size, size_t nmemb, FILE *fp);

The **explain_message_fread** function may be used to obtain an explanation of an error returned by the *fread*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
size_t how_many = fread(ptr, size, nmemb, fp);
if (how_many == 0 && ferror(fp))
{
    char message[3000];
    explain_message_fread(message, sizeof(message), ptr, size, nmemb, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*ptr* The original ptr, exactly as passed to the *fread*(3) system call.

*size* The original size, exactly as passed to the *fread*(3) system call.

*nmemb* The original nmemb, exactly as passed to the *fread*(3) system call.

*fp* The original fp, exactly as passed to the *fread*(3) system call.

**explain_message_errno_fread**

void explain_message_errno_fread(char *message, int message_size, int errnum, void *ptr, size_t size, size_t nmemb, FILE *fp);

The **explain_message_errno_fread** function may be used to obtain an explanation of an error returned by the *fread*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
size_t how_many = fread(ptr, size, nmemb, fp);
if (how_many == 0 && ferror(fp))
{
    int err = errno;
```

```
        char message[3000];
        explain_message_errno_fread(message, sizeof(message), err,
            ptr, size, nmemb, fp);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ptr* The original ptr, exactly as passed to the *fread*(3) system call.

*size* The original size, exactly as passed to the *fread*(3) system call.

*nmemb* The original nmemb, exactly as passed to the *fread*(3) system call.

*fp* The original fp, exactly as passed to the *fread*(3) system call.

## SEE ALSO

*fread*(3) binary stream input

*explain_fread_or_die*(3)
binary stream input and report errors

## COPYRIGHT

## NAME

explain_fread_or_die − binary stream input and report errors

## SYNOPSIS

#include <libexplain/fread.h>

void explain_fread_or_die(void *ptr, size_t size, size_t nmemb, FILE *fp);

## DESCRIPTION

The **explain_fread_or_die** function is used to call the *fread*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fread*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
size_t how_many = explain_fread_or_die(ptr, size, nmemb, fp);
```

*ptr* The ptr, exactly as to be passed to the *fread*(3) system call.

*size* The size, exactly as to be passed to the *fread*(3) system call.

*nmemb* The nmemb, exactly as to be passed to the *fread*(3) system call.

*fp* The fp, exactly as to be passed to the *fread*(3) system call.

Returns: This function only returns on success, the number read or 0 on end-of-input. On failure, prints an explanation and exits.

## SEE ALSO

*fread*(3) binary stream input

*explain_fread*(3)
explain *fread*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_freopen − explain freopen(3) errors

## SYNOPSIS

#include <libexplain/freopen.h>
const char *explain_freopen(const char *pathname, const char *flags, FILE *fp);
const char *explain_errno_freopen(int errnum, const char *pathname, const char *flags, FILE *fp);
void explain_message_freopen(char *message, int message_size, const char *pathname, const char *flags, FILE *fp);
void explain_message_errno_freopen(char *message, int message_size, int errnum, const char *pathname, const char *flags, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for *freopen*(3) errors.

### explain_freopen

const char *explain_freopen(const char *pathname, const char *flags, FILE *fp);

The explain_freopen function is used to obtain an explanation of an error returned by the *freopen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (!freopen(pathname, flags, fp))
{
    fprintf(stderr, '%s0, explain_freopen(pathname, flags, fp));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *freopen*(3) system call.

*flags*
: The original flags, exactly as passed to the *freopen*(3) system call.

*fp*
: The original fp, exactly as passed to the *freopen*(3) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_freopen

const char *explain_errno_freopen(int errnum, const char *pathname, const char *flags, FILE *fp);

The explain_errno_freopen function is used to obtain an explanation of an error returned by the *freopen*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (freopen(pathname, flags, fp))
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_freopen(err, pathname,
        flags, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *freopen*(3) system call.

*flags* The original flags, exactly as passed to the *freopen*(3) system call.

*fp* The original fp, exactly as passed to the *freopen*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_freopen**

void explain_message_freopen(char *message, int message_size, const char *pathname, const char *flags, FILE *fp);

The explain_message_freopen function is used to obtain an explanation of an error returned by the *freopen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (!freopen(pathname, flags, fp))
{
    char message[3000];
    explain_message_freopen(message, sizeof(message), pathname, flags,
        fp);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *freopen*(3) system call.

*flags* The original flags, exactly as passed to the *freopen*(3) system call.

*fp* The original fp, exactly as passed to the *freopen*(3) system call.

**explain_message_errno_freopen**

void explain_message_errno_freopen(char *message, int message_size, int errnum, const char *pathname, const char *flags, FILE *fp);

The explain_message_errno_freopen function is used to obtain an explanation of an error returned by the *freopen*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (!freopen(pathname, flags, fp))
{
    int err = errno;
    char message[3000];
    explain_message_errno_freopen(message, sizeof(message), err,
```

```
            pathname, flags, fp);
        fprintf(stderr, '%s0, message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *freopen*(3) system call.

*flags* The original flags, exactly as passed to the *freopen*(3) system call.

*fp* The original fp, exactly as passed to the *freopen*(3) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_freopen_or_die − open file and report errors

## SYNOPSIS

#include <libexplain/freopen.h>
void explain_freopen_or_die(const char *pathname, const char *flags, FILE *fp);

## DESCRIPTION

The explain_freopen_or_die function is used to reopen a file via the *freopen*(3) system call. On failure it will print an explanation, obtained from the *linexplain_freopen*(3) function, on the standard error stream and then exit.

This function is intended to be used in a fashion similar to the following example:

```
explain_freopen_or_die(pathname, flags, fp);
```

*pathname*
: The pathname, exactly as to be passed to the *freopen*(3) system call.

*flags*
: The flags, exactly as to be passed to the *freopen*(3) system call.

*fp*
: The fp, exactly as to be passed to the *freopen*(3) system call.

Returns: Only ever return on success. Never returns on failure.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_fseek − explain *fseek*(3) errors

## SYNOPSIS

#include <libexplain/fseek.h>

const char *explain_fseek(FILE *fp, long offset, int whence);
const char *explain_errno_fseek(int errnum, FILE *fp, long offset, int whence);
void explain_message_fseek(char *message, int message_size, FILE *fp, long offset, int whence);
void explain_message_errno_fseek(char *message, int message_size, int errnum, FILE *fp, long offset, int whence);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fseek*(3) system call.

### explain_fseek

const char *explain_fseek(FILE *fp, long offset, int whence);

The **explain_fseek** function is used to obtain an explanation of an error returned by the *fseek*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *fseek*(3) system call.

*offset* The original offset, exactly as passed to the *fseek*(3) system call.

*whence* The original whence, exactly as passed to the *fseek*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fseek(fp, offset, whence) < 0)
{
   fprintf(stderr, "%s\n", explain_fseek(fp, offset, whence));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fseek_or_die*(3) function.

### explain_errno_fseek

const char *explain_errno_fseek(int errnum, FILE *fp, long offset, int whence);

The **explain_errno_fseek** function is used to obtain an explanation of an error returned by the *fseek*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fseek*(3) system call.

*offset* The original offset, exactly as passed to the *fseek*(3) system call.

*whence* The original whence, exactly as passed to the *fseek*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fseek(fp, offset, whence) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_fseek(err, fp, offset,
   whence));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fseek_or_die*(3) function.

**explain_message_fseek**

void explain_message_fseek(char *message, int message_size, FILE *fp, long offset, int whence);

The **explain_message_fseek** function is used to obtain an explanation of an error returned by the *fseek*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fseek*(3) system call.

*offset* The original offset, exactly as passed to the *fseek*(3) system call.

*whence* The original whence, exactly as passed to the *fseek*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fseek(fp, offset, whence) < 0)
{
    char message[3000];
   explain_message_fseek(message, sizeof(message), fp, offset,
   whence);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fseek_or_die*(3) function.

**explain_message_errno_fseek**

void explain_message_errno_fseek(char *message, int message_size, int errnum, FILE *fp, long offset, int whence);

The **explain_message_errno_fseek** function is used to obtain an explanation of an error returned by the *fseek*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fseek*(3) system call.

*offset* The original offset, exactly as passed to the *fseek*(3) system call.

*whence* The original whence, exactly as passed to the *fseek*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fseek(fp, offset, whence) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_fseek(message, sizeof(message), err, fp,
   offset, whence);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fseek_or_die*(3) function.

## SEE ALSO

*fseek*(3) reposition a stream

*explain_fseek_or_die*(3)
reposition a stream and report errors

## COPYRIGHT

## NAME

explain_fseek_or_die − reposition a stream and report errors

## SYNOPSIS

#include <libexplain/fseek.h>

void explain_fseek_or_die(FILE *fp, long offset, int whence);
int explain_fseek_on_error(FILE *fp, long offset, int whence);

## DESCRIPTION

The **explain_fseek_or_die** function is used to call the *fseek*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fseek*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fseek_on_error** function is used to call the *fseek*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fseek*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *fseek*(3) system call.

*offset* The offset, exactly as to be passed to the *fseek*(3) system call.

*whence* The whence, exactly as to be passed to the *fseek*(3) system call.

## RETURN VALUE

The **explain_fseek_or_die** function only returns on success, see *fseek*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fseek_on_error** function always returns the value return by the wrapped *fseek*(3) system call.

## EXAMPLE

The **explain_fseek_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fseek_or_die(fp, offset, whence);
```

## SEE ALSO

*fseek*(3) reposition a stream

*explain_fseek*(3)
explain *fseek*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fsetpos − explain *fsetpos*(3) errors

## SYNOPSIS

#include <libexplain/fsetpos.h>

const char *explain_fsetpos(FILE *fp, fpos_t *pos);
const char *explain_errno_fsetpos(int errnum, FILE *fp, fpos_t *pos);
void explain_message_fsetpos(char *message, int message_size, FILE *fp, fpos_t *pos);
void explain_message_errno_fsetpos(char *message, int message_size, int errnum, FILE *fp, fpos_t *pos);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fsetpos*(3) system call.

### explain_fsetpos

const char *explain_fsetpos(FILE *fp, fpos_t *pos);

The **explain_fsetpos** function is used to obtain an explanation of an error returned by the *fsetpos*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *fsetpos*(3) system call.

*pos* The original pos, exactly as passed to the *fsetpos*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fsetpos(fp, pos) < 0)
{
   fprintf(stderr, "%s\n", explain_fsetpos(fp, pos));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fsetpos_or_die*(3) function.

### explain_errno_fsetpos

const char *explain_errno_fsetpos(int errnum, FILE *fp, fpos_t *pos);

The **explain_errno_fsetpos** function is used to obtain an explanation of an error returned by the *fsetpos*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fsetpos*(3) system call.

*pos* The original pos, exactly as passed to the *fsetpos*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fsetpos(fp, pos) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fsetpos(err, fp, pos));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fsetpos_or_die*(3) function.

**explain_message_fsetpos**

void explain_message_fsetpos(char *message, int message_size, FILE *fp, fpos_t *pos);

The **explain_message_fsetpos** function is used to obtain an explanation of an error returned by the *fsetpos*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *fsetpos*(3) system call.

*pos* The original pos, exactly as passed to the *fsetpos*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fsetpos(fp, pos) < 0)
{
    char message[3000];
    explain_message_fsetpos(message, sizeof(message), fp, pos);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fsetpos_or_die*(3) function.

**explain_message_errno_fsetpos**

void explain_message_errno_fsetpos(char *message, int message_size, int errnum, FILE *fp, fpos_t *pos);

The **explain_message_errno_fsetpos** function is used to obtain an explanation of an error returned by the *fsetpos*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *fsetpos*(3) system call.

*pos* The original pos, exactly as passed to the *fsetpos*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fsetpos(fp, pos) < 0)
{
    int err = errno;
    char message[3000];
```

```
        explain_message_errno_fsetpos(message, sizeof(message), err,
        fp, pos);
         fprintf(stderr, "%s\n", message);
         exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_fsetpos_or_die*(3) function.

## SEE ALSO

*fsetpos*(3)
reposition a stream

*explain_fsetpos_or_die*(3)
reposition a stream and report errors

## COPYRIGHT

## NAME

explain_fsetpos_or_die − reposition a stream and report errors

## SYNOPSIS

#include <libexplain/fsetpos.h>

void explain_fsetpos_or_die(FILE *fp, fpos_t *pos);
int explain_fsetpos_on_error(FILE *fp, fpos_t *pos);

## DESCRIPTION

The **explain_fsetpos_or_die** function is used to call the *fsetpos*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fsetpos*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fsetpos_on_error** function is used to call the *fsetpos*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fsetpos*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *fsetpos*(3) system call.

*pos* The pos, exactly as to be passed to the *fsetpos*(3) system call.

## RETURN VALUE

The **explain_fsetpos_or_die** function only returns on success, see *fsetpos*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fsetpos_on_error** function always returns the value return by the wrapped *fsetpos*(3) system call.

## EXAMPLE

The **explain_fsetpos_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fsetpos_or_die(fp, pos);
```

## SEE ALSO

*fsetpos*(3)
reposition a stream

*explain_fsetpos*(3)
explain *fsetpos*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fstat − explain fstat(2) errors

## SYNOPSIS

#include <libexplain/fstat.h>

const char *explain_fstat(int fildes, struct stat *buf);
const char *explain_errno_fstat(int errnum, int fildes, struct stat *buf);
void explain_message_fstat(char *message, int message_size, int fildes, struct stat *buf);
void explain_message_errno_fstat(char *message, int message_size, int errnum, int fildes, struct stat *buf);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fstat*(2) system call.

### explain_fstat

const char *explain_fstat(int fildes, struct stat *buf);

The **explain_fstat** function is used to obtain an explanation of an error returned by the *fstat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fstat(fildes, buf) < 0)
{
    fprintf(stderr, "%s\n", explain_fstat(fildes, buf));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *fstat*(2) system call.

*buf* The original buf, exactly as passed to the *fstat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fstat

const char *explain_errno_fstat(int errnum, int fildes, struct stat *buf);

The **explain_errno_fstat** function is used to obtain an explanation of an error returned by the *fstat*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fstat(fildes, buf) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fstat(err, fildes, buf));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fstat*(2) system call.

*buf* The original buf, exactly as passed to the *fstat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fstat**

void explain_message_fstat(char *message, int message_size, int fildes, struct stat *buf);

The **explain_message_fstat** function may be used to obtain an explanation of an error returned by the *fstat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fstat(fildes, buf) < 0)
{
    char message[3000];
    explain_message_fstat(message, sizeof(message), fildes, buf);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fstat*(2) system call.

*buf* The original buf, exactly as passed to the *fstat*(2) system call.

**explain_message_errno_fstat**

void explain_message_errno_fstat(char *message, int message_size, int errnum, int fildes, struct stat *buf);

The **explain_message_errno_fstat** function may be used to obtain an explanation of an error returned by the *fstat*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fstat(fildes, buf) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fstat(message, sizeof(message), err, fildes, buf);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fstat*(2) system call.

*buf* The original buf, exactly as passed to the *fstat*(2) system call.

## SEE ALSO

*fstat*(2) get file status

*explain_fstat_or_die*(3)
get file status and report errors

## COPYRIGHT

## NAME

explain_fstatfs − explain fstatfs(2) errors

## SYNOPSIS

#include <libexplain/fstatfs.h>

const char *explain_fstatfs(int fildes, struct statfs *data);
const char *explain_errno_fstatfs(int errnum, int fildes, struct statfs *data);
void explain_message_fstatfs(char *message, int message_size, int fildes, struct statfs *data);
void explain_message_errno_fstatfs(char *message, int message_size, int errnum, int fildes, struct statfs *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fstatfs*(2) system call.

### explain_fstatfs

const char *explain_fstatfs(int fildes, struct statfs *data);

The **explain_fstatfs** function is used to obtain an explanation of an error returned by the *fstatfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *fstatfs*(2) system call.

*data* The original data, exactly as passed to the *fstatfs*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fstatfs(fildes, data) < 0)
{
   fprintf(stderr, "%s\n", explain_fstatfs(fildes, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fstatfs_or_die*(3) function.

### explain_errno_fstatfs

const char *explain_errno_fstatfs(int errnum, int fildes, struct statfs *data);

The **explain_errno_fstatfs** function is used to obtain an explanation of an error returned by the *fstatfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fstatfs*(2) system call.

*data* The original data, exactly as passed to the *fstatfs*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fstatfs(fildes, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_fstatfs(err, fildes,
   data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fstatfs_or_die*(3) function.

**explain_message_fstatfs**

void explain_message_fstatfs(char *message, int message_size, int fildes, struct statfs *data);

The **explain_message_fstatfs** function is used to obtain an explanation of an error returned by the *fstatfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fstatfs*(2) system call.

*data* The original data, exactly as passed to the *fstatfs*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fstatfs(fildes, data) < 0)
{
    char message[3000];
   explain_message_fstatfs(message, sizeof(message), fildes,
   data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fstatfs_or_die*(3) function.

**explain_message_errno_fstatfs**

void explain_message_errno_fstatfs(char *message, int message_size, int errnum, int fildes, struct statfs *data);

The **explain_message_errno_fstatfs** function is used to obtain an explanation of an error returned by the *fstatfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fstatfs*(2) system call.

*data* The original data, exactly as passed to the *fstatfs*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fstatfs(fildes, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_fstatfs(message, sizeof(message), err,
   fildes, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fstatfs_or_die*(3) function.

## SEE ALSO

*fstatfs*(2)
get file system statistics

*explain_fstatfs_or_die*(3)
get file system statistics and report errors

## COPYRIGHT

## NAME

explain_fstatfs_or_die − get file system statistics and report errors

## SYNOPSIS

#include <libexplain/fstatfs.h>

void explain_fstatfs_or_die(int fildes, struct statfs *data);
int explain_fstatfs_on_error(int fildes, struct statfs *data);

## DESCRIPTION

The **explain_fstatfs_or_die** function is used to call the *fstatfs*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fstatfs*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fstatfs_on_error** function is used to call the *fstatfs*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fstatfs*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *fstatfs*(2) system call.

*data* The data, exactly as to be passed to the *fstatfs*(2) system call.

## RETURN VALUE

The **explain_fstatfs_or_die** function only returns on success, see *fstatfs*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fstatfs_on_error** function always returns the value return by the wrapped *fstatfs*(2) system call.

## EXAMPLE

The **explain_fstatfs_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fstatfs_or_die(fildes, data);
```

## SEE ALSO

*fstatfs*(2)
get file system statistics

*explain_fstatfs*(3)
explain *fstatfs*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fstat_or_die − get file status and report errors

## SYNOPSIS

#include <libexplain/fstat.h>

void explain_fstat_or_die(int fildes, struct stat *buf);

## DESCRIPTION

The **explain_fstat_or_die** function is used to call the *fstat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fstat*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_fstat_or_die(fildes, buf);
```

*fildes* The fildes, exactly as to be passed to the *fstat*(2) system call.

*buf* The buf, exactly as to be passed to the *fstat*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*fstat*(2) get file status

*explain_fstat*(3)
explain *fstat*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fstatvfs − explain *fstatvfs*(2) errors

## SYNOPSIS

#include <libexplain/fstatvfs.h>

const char *explain_fstatvfs(int fildes, struct statvfs *data);
const char *explain_errno_fstatvfs(int errnum, int fildes, struct statvfs *data);
void explain_message_fstatvfs(char *message, int message_size, int fildes, struct statvfs *data);
void explain_message_errno_fstatvfs(char *message, int message_size, int errnum, int fildes, struct statvfs *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fstatvfs*(2) system call.

### explain_fstatvfs

const char *explain_fstatvfs(int fildes, struct statvfs *data);

The **explain_fstatvfs** function is used to obtain an explanation of an error returned by the *fstatvfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *fstatvfs*(2) system call.

*data* The original data, exactly as passed to the *fstatvfs*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fstatvfs(fildes, data) < 0)
{
   fprintf(stderr, "%s\n", explain_fstatvfs(fildes, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fstatvfs_or_die*(3) function.

### explain_errno_fstatvfs

const char *explain_errno_fstatvfs(int errnum, int fildes, struct statvfs *data);

The **explain_errno_fstatvfs** function is used to obtain an explanation of an error returned by the *fstatvfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fstatvfs*(2) system call.

*data* The original data, exactly as passed to the *fstatvfs*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fstatvfs(fildes, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_fstatvfs(err, fildes,
   data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fstatvfs_or_die*(3) function.

**explain_message_fstatvfs**

void explain_message_fstatvfs(char *message, int message_size, int fildes, struct statvfs *data);

The **explain_message_fstatvfs** function is used to obtain an explanation of an error returned by the *fstatvfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fstatvfs*(2) system call.

*data* The original data, exactly as passed to the *fstatvfs*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fstatvfs(fildes, data) < 0)
{
    char message[3000];
   explain_message_fstatvfs(message, sizeof(message), fildes,
   data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fstatvfs_or_die*(3) function.

**explain_message_errno_fstatvfs**

void explain_message_errno_fstatvfs(char *message, int message_size, int errnum, int fildes, struct statvfs *data);

The **explain_message_errno_fstatvfs** function is used to obtain an explanation of an error returned by the *fstatvfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fstatvfs*(2) system call.

*data* The original data, exactly as passed to the *fstatvfs*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fstatvfs(fildes, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_fstatvfs(message, sizeof(message), err,
   fildes, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fstatvfs_or_die*(3) function.

## SEE ALSO

*fstatvfs*(2)
get file system statistics

*explain_fstatvfs_or_die*(3)
get file system statistics and report errors

## COPYRIGHT

## NAME

explain_fstatvfs_or_die − get file system statistics and report errors

## SYNOPSIS

#include <libexplain/fstatvfs.h>

void explain_fstatvfs_or_die(int fildes, struct statvfs *data);
int explain_fstatvfs_on_error(int fildes, struct statvfs *data);

## DESCRIPTION

The **explain_fstatvfs_or_die** function is used to call the *fstatvfs*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fstatvfs*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fstatvfs_on_error** function is used to call the *fstatvfs*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fstatvfs*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *fstatvfs*(2) system call.

*data* The data, exactly as to be passed to the *fstatvfs*(2) system call.

## RETURN VALUE

The **explain_fstatvfs_or_die** function only returns on success, see *fstatvfs*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fstatvfs_on_error** function always returns the value return by the wrapped *fstatvfs*(2) system call.

## EXAMPLE

The **explain_fstatvfs_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fstatvfs_or_die(fildes, data);
```

## SEE ALSO

*fstatvfs*(2)
get file system statistics

*explain_fstatvfs*(3)
explain *fstatvfs*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fsync − explain *fsync*(2) errors

## SYNOPSIS

#include <libexplain/fsync.h>

const char *explain_fsync(int fildes);
const char *explain_errno_fsync(int errnum, int fildes);
void explain_message_fsync(char *message, int message_size, int fildes);
void explain_message_errno_fsync(char *message, int message_size, int errnum, int fildes);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fsync*(2) system call.

### explain_fsync

const char *explain_fsync(int fildes);

The **explain_fsync** function is used to obtain an explanation of an error returned by the *fsync*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *fsync*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fsync(fildes) < 0)
{
   fprintf(stderr, "%s\n", explain_fsync(fildes));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fsync_or_die*(3) function.

### explain_errno_fsync

const char *explain_errno_fsync(int errnum, int fildes);

The **explain_errno_fsync** function is used to obtain an explanation of an error returned by the *fsync*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fsync*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fsync(fildes) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fsync(err, fildes));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fsync_or_die*(3) function.

**explain_message_fsync**

void explain_message_fsync(char *message, int message_size, int fildes);

The **explain_message_fsync** function is used to obtain an explanation of an error returned by the *fsync*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *fsync*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fsync(fildes) < 0)
{
    char message[3000];
    explain_message_fsync(message, sizeof(message), fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fsync_or_die*(3) function.

**explain_message_errno_fsync**

void explain_message_errno_fsync(char *message, int message_size, int errnum, int fildes);

The **explain_message_errno_fsync** function is used to obtain an explanation of an error returned by the *fsync*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *fsync*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (fsync(fildes) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fsync(message, sizeof(message), err,
    fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_fsync_or_die*(3) function.

## SEE ALSO

*fsync*(2) synchronize a file's in-core state with storage device

*explain_fsync_or_die*(3)
synchronize a file's in-core state with storage device and report errors

## COPYRIGHT

## NAME

explain_fsync_or_die − synchronize a file with storage device and report errors

## SYNOPSIS

#include <libexplain/fsync.h>

void explain_fsync_or_die(int fildes);
int explain_fsync_on_error(int fildes);

## DESCRIPTION

The **explain_fsync_or_die** function is used to call the *fsync*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fsync*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_fsync_on_error** function is used to call the *fsync*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_fsync*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *fsync*(2) system call.

## RETURN VALUE

The **explain_fsync_or_die** function only returns on success, see *fsync*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_fsync_on_error** function always returns the value return by the wrapped *fsync*(2) system call.

## EXAMPLE

The **explain_fsync_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_fsync_or_die(fildes);
```

## SEE ALSO

*fsync*(2) synchronize a file's in-core state with storage device

*explain_fsync*(3)
explain *fsync*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_ftell − explain *ftell*(3) errors

## SYNOPSIS

#include <libexplain/ftell.h>

const char *explain_ftell(FILE *fp);
const char *explain_errno_ftell(int errnum, FILE *fp);
void explain_message_ftell(char *message, int message_size, FILE *fp);
void explain_message_errno_ftell(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *ftell*(3) system call.

### explain_ftell

const char *explain_ftell(FILE *fp);

The **explain_ftell** function is used to obtain an explanation of an error returned by the *ftell*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *ftell*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = ftell(fp);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_ftell(fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ftell_or_die*(3) function.

### explain_errno_ftell

const char *explain_errno_ftell(int errnum, FILE *fp);

The **explain_errno_ftell** function is used to obtain an explanation of an error returned by the *ftell*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *ftell*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = ftell(fp);
```

```
if (result < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_ftell(err, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ftell_or_die*(3) function.

**explain_message_ftell**

void explain_message_ftell(char *message, int message_size, FILE *fp);

The **explain_message_ftell** function is used to obtain an explanation of an error returned by the *ftell*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *ftell*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = ftell(fp);
if (result < 0)
{
    char message[3000];
    explain_message_ftell(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ftell_or_die*(3) function.

**explain_message_errno_ftell**

void explain_message_errno_ftell(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_ftell** function is used to obtain an explanation of an error returned by the *ftell*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *ftell*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = ftell(fp);
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_ftell(message, sizeof(message), err,
```

```
                fp);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_ftell_or_die*(3) function.

## SEE ALSO

*ftell*(3) reposition a stream

*explain_ftell_or_die*(3)
reposition a stream and report errors

## COPYRIGHT

## NAME

explain_ftell_or_die − get stream position and report errors

## SYNOPSIS

#include <libexplain/ftell.h>

long explain_ftell_or_die(FILE *fp);
long explain_ftell_on_error(FILE *fp);

## DESCRIPTION

The **explain_ftell_or_die** function is used to call the *ftell*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ftell*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_ftell_on_error** function is used to call the *ftell*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ftell*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *ftell*(3) system call.

## RETURN VALUE

The **explain_ftell_or_die** function only returns on success, see *ftell*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_ftell_on_error** function always returns the value return by the wrapped *ftell*(3) system call.

## EXAMPLE

The **explain_ftell_or_die** function is intended to be used in a fashion similar to the following example:

```
long result = explain_ftell_or_die(fp);
```

## SEE ALSO

*ftell*(3) get stream position

*explain_ftell*(3)
explain *ftell*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_ftime − explain ftime(3) errors

## SYNOPSIS

#include <libexplain/ftime.h>

const char *explain_ftime(struct timeb *tp);
const char *explain_errno_ftime(int errnum, struct timeb *tp);
void explain_message_ftime(char *message, int message_size, struct timeb *tp);
void explain_message_errno_ftime(char *message, int message_size, int errnum, struct timeb *tp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *ftime*(3) system call.

### explain_ftime

const char *explain_ftime(struct timeb *tp);

The **explain_ftime** function is used to obtain an explanation of an error returned by the *ftime*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*tp* The original tp, exactly as passed to the *ftime*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ftime(tp) < 0)
{
   fprintf(stderr, "%s\n", explain_ftime(tp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ftime_or_die*(3) function.

### explain_errno_ftime

const char *explain_errno_ftime(int errnum, struct timeb *tp);

The **explain_errno_ftime** function is used to obtain an explanation of an error returned by the *ftime*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*tp* The original tp, exactly as passed to the *ftime*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ftime(tp) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_ftime(err, tp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ftime_or_die*(3) function.

**explain_message_ftime**

void explain_message_ftime(char *message, int message_size, struct timeb *tp);

The **explain_message_ftime** function is used to obtain an explanation of an error returned by the *ftime*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*tp* The original tp, exactly as passed to the *ftime*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ftime(tp) < 0)
{
    char message[3000];
    explain_message_ftime(message, sizeof(message), tp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ftime_or_die*(3) function.

**explain_message_errno_ftime**

void explain_message_errno_ftime(char *message, int message_size, int errnum, struct timeb *tp);

The **explain_message_errno_ftime** function is used to obtain an explanation of an error returned by the *ftime*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*tp* The original tp, exactly as passed to the *ftime*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ftime(tp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_ftime(message, sizeof(message), err,
    tp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ftime_or_die*(3) function.

## SEE ALSO

*ftime*(3) return date and time

*explain_ftime_or_die*(3)
return date and time and report errors

## COPYRIGHT

## NAME

explain_ftime_or_die − return date and time and report errors

## SYNOPSIS

#include <libexplain/ftime.h>

void explain_ftime_or_die(struct timeb *tp);
int explain_ftime_on_error(struct timeb *tp);

## DESCRIPTION

The **explain_ftime_or_die** function is used to call the *ftime*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ftime*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_ftime_on_error** function is used to call the *ftime*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ftime*(3) function, but still returns to the caller.

*tp* The tp, exactly as to be passed to the *ftime*(3) system call.

## RETURN VALUE

The **explain_ftime_or_die** function only returns on success, see *ftime*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_ftime_on_error** function always returns the value return by the wrapped *ftime*(3) system call.

## EXAMPLE

The **explain_ftime_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_ftime_or_die(tp);
```

## SEE ALSO

*ftime*(3) return date and time

*explain_ftime*(3)
explain *ftime*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_ftruncate − explain ftruncate(2) errors

## SYNOPSIS

#include <libexplain/ftruncate.h>

const char *explain_ftruncate(int fildes, long long length);
const char *explain_errno_ftruncate(int errnum, int fildes, long long length);
void explain_message_ftruncate(char *message, int message_size, int fildes, long long length);
void explain_message_errno_ftruncate(char *message, int message_size, int errnum, int fildes, long long length);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *ftruncate*(2) system call.

### explain_ftruncate

const char *explain_ftruncate(int fildes, long long length);

The **explain_ftruncate** function is used to obtain an explanation of an error returned by the *ftruncate*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (ftruncate(fildes, length) < 0)
{
    fprintf(stderr, "%s\n", explain_ftruncate(fildes, length));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *ftruncate*(2) system call.

*length* The original length, exactly as passed to the *ftruncate*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_ftruncate

const char *explain_errno_ftruncate(int errnum, int fildes, long long length);

The **explain_errno_ftruncate** function is used to obtain an explanation of an error returned by the *ftruncate*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (ftruncate(fildes, length) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_ftruncate(err, fildes, length));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *ftruncate*(2) system call.

*length* The original length, exactly as passed to the *ftruncate*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_ftruncate**

void explain_message_ftruncate(char *message, int message_size, int fildes, long long length);

The **explain_message_ftruncate** function may be used to obtain an explanation of an error returned by the *ftruncate*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (ftruncate(fildes, length) < 0)
{
    char message[3000];
    explain_message_ftruncate(message, sizeof(message), fildes, length);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *ftruncate*(2) system call.

*length* The original length, exactly as passed to the *ftruncate*(2) system call.

**explain_message_errno_ftruncate**

void explain_message_errno_ftruncate(char *message, int message_size, int errnum, int fildes, long long length);

The **explain_message_errno_ftruncate** function may be used to obtain an explanation of an error returned by the *ftruncate*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (ftruncate(fildes, length) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_ftruncate(message, sizeof(message), err,
        fildes, length);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *ftruncate*(2) system call.

*length* The original length, exactly as passed to the *ftruncate*(2) system call.

## SEE ALSO

*ftruncate*(2)
truncate a file to a specified length

*explain_ftruncate_or_die*(3)
truncate a file and report errors

## COPYRIGHT

## NAME

explain_ftruncate_or_die − truncate a file and report errors

## SYNOPSIS

#include <libexplain/ftruncate.h>

void explain_ftruncate_or_die(int fildes, long long length);

## DESCRIPTION

The **explain_ftruncate_or_die** function is used to call the *ftruncate*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_ftruncate*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_ftruncate_or_die(fildes, length);
```

*fildes* The fildes, exactly as to be passed to the *ftruncate*(2) system call.

*length* The length, exactly as to be passed to the *ftruncate*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*ftruncate*(2)
truncate a file to a specified length

*explain_ftruncate*(3)
explain *ftruncate*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_futimes − explain futimes(3) errors

## SYNOPSIS

#include <libexplain/futimes.h>

const char *explain_futimes(int fildes, const struct timeval *tv);
const char *explain_errno_futimes(int errnum, int fildes, const struct timeval *tv);
void explain_message_futimes(char *message, int message_size, int fildes, const struct timeval *tv);
void explain_message_errno_futimes(char *message, int message_size, int errnum, int fildes, const struct timeval *tv);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *futimes*(3) system call.

### explain_futimes

const char *explain_futimes(int fildes, const struct timeval *tv);

The **explain_futimes** function is used to obtain an explanation of an error returned by the *futimes*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (futimes(fildes, tv) < 0)
{
    fprintf(stderr, "%s\n", explain_futimes(fildes, tv));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_futimes_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *futimes*(3) system call.

*tv* The original tv, exactly as passed to the *futimes*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_futimes

const char *explain_errno_futimes(int errnum, int fildes, const struct timeval *tv);

The **explain_errno_futimes** function is used to obtain an explanation of an error returned by the *futimes*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (futimes(fildes, tv) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_futimes(err, fildes, tv));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_futimes_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *futimes*(3) system call.

*tv* The original tv, exactly as passed to the *futimes*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_futimes**

void explain_message_futimes(char *message, int message_size, int fildes, const struct timeval *tv);

The **explain_message_futimes** function may be used to obtain an explanation of an error returned by the *futimes*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (futimes(fildes, tv) < 0)
{
    char message[3000];
    explain_message_futimes(message, sizeof(message), fildes, tv);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_futimes_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *futimes*(3) system call.

*tv* The original tv, exactly as passed to the *futimes*(3) system call.

**explain_message_errno_futimes**

void explain_message_errno_futimes(char *message, int message_size, int errnum, int fildes, const struct timeval *tv);

The **explain_message_errno_futimes** function may be used to obtain an explanation of an error returned by the *futimes*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (futimes(fildes, tv) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_futimes(message, sizeof(message), err, fildes, tv);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_futimes_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *futimes*(3) system call.

*tv* The original tv, exactly as passed to the *futimes*(3) system call.

## SEE ALSO

*futimes*(3)
change file timestamps

*explain_futimes_or_die*(3)
change file timestamps and report errors

## COPYRIGHT

## NAME

explain_futimes_or_die − change file timestamps and report errors

## SYNOPSIS

#include <libexplain/futimes.h>

void explain_futimes_or_die(int fildes, const struct timeval *tv);

## DESCRIPTION

The **explain_futimes_or_die** function is used to call the *futimes*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_futimes*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_futimes_or_die(fildes, tv);
```

*fildes* The fildes, exactly as to be passed to the *futimes*(3) system call.

*tv* The tv, exactly as to be passed to the *futimes*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*futimes*(3)
change file timestamps

*explain_futimes*(3)
explain *futimes*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_fwrite − explain fwrite(3) errors

## SYNOPSIS

#include <libexplain/fwrite.h>

const char *explain_fwrite(const void *ptr, size_t size, size_t nmemb, FILE *fp);
const char *explain_errno_fwrite(int errnum, const void *ptr, size_t size, size_t nmemb, FILE *fp);
void explain_message_fwrite(char *message, int message_size, const void *ptr, size_t size, size_t nmemb, FILE *fp);
void explain_message_errno_fwrite(char *message, int message_size, int errnum, const void *ptr, size_t size, size_t nmemb, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *fwrite*(3) system call.

### explain_fwrite

const char *explain_fwrite(const void *ptr, size_t size, size_t nmemb, FILE *fp);

The **explain_fwrite** function is used to obtain an explanation of an error returned by the *fwrite*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fwrite(ptr, size, nmemb, fp) < 0)
{
    fprintf(stderr, "%s\n", explain_fwrite(ptr, size, nmemb, fp));
    exit(EXIT_FAILURE);
}
```

*ptr* The original ptr, exactly as passed to the *fwrite*(3) system call.

*size* The original size, exactly as passed to the *fwrite*(3) system call.

*nmemb* The original nmemb, exactly as passed to the *fwrite*(3) system call.

*fp* The original fp, exactly as passed to the *fwrite*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_fwrite

const char *explain_errno_fwrite(int errnum, const void *ptr, size_t size, size_t nmemb, FILE *fp);

The **explain_errno_fwrite** function is used to obtain an explanation of an error returned by the *fwrite*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fwrite(ptr, size, nmemb, fp) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_fwrite(err,
        ptr, size, nmemb, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ptr* The original ptr, exactly as passed to the *fwrite*(3) system call.

*size* The original size, exactly as passed to the *fwrite*(3) system call.

*nmemb* The original nmemb, exactly as passed to the *fwrite*(3) system call.

*fp* The original fp, exactly as passed to the *fwrite*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_fwrite**

void explain_message_fwrite(char *message, int message_size, const void *ptr, size_t size, size_t nmemb, FILE *fp);

The **explain_message_fwrite** function may be used to obtain an explanation of an error returned by the *fwrite*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (fwrite(ptr, size, nmemb, fp) < 0)
{
    char message[3000];
    explain_message_fwrite(message, sizeof(message), ptr, size, nmemb, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*ptr* The original ptr, exactly as passed to the *fwrite*(3) system call.

*size* The original size, exactly as passed to the *fwrite*(3) system call.

*nmemb* The original nmemb, exactly as passed to the *fwrite*(3) system call.

*fp* The original fp, exactly as passed to the *fwrite*(3) system call.

**explain_message_errno_fwrite**

void explain_message_errno_fwrite(char *message, int message_size, int errnum, const void *ptr, size_t size, size_t nmemb, FILE *fp);

The **explain_message_errno_fwrite** function may be used to obtain an explanation of an error returned by the *fwrite*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (fwrite(ptr, size, nmemb, fp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_fwrite(message, sizeof(message), err,
```

```
            ptr, size, nmemb, fp);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ptr* The original ptr, exactly as passed to the *fwrite*(3) system call.

*size* The original size, exactly as passed to the *fwrite*(3) system call.

*nmemb* The original nmemb, exactly as passed to the *fwrite*(3) system call.

*fp* The original fp, exactly as passed to the *fwrite*(3) system call.

## SEE ALSO

*fwrite*(3) binary stream output

*explain_fwrite_or_die*(3)
binary stream output and report errors

## COPYRIGHT

## NAME

explain_fwrite_or_die − binary stream output and report errors

## SYNOPSIS

#include <libexplain/fwrite.h>

size_t explain_fwrite_or_die(const void *ptr, size_t size, size_t nmemb, FILE *fp);

## DESCRIPTION

The **explain_fwrite_or_die** function is used to call the *fwrite*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_fwrite*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
size_t result = explain_fwrite_or_die(ptr, size, nmemb, fp);
```

*ptr* The ptr, exactly as to be passed to the *fwrite*(3) system call.

*size* The size, exactly as to be passed to the *fwrite*(3) system call.

*nmemb* The nmemb, exactly as to be passed to the *fwrite*(3) system call.

*fp* The fp, exactly as to be passed to the *fwrite*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*fwrite*(3) binary stream output

*explain_fwrite*(3)
explain *fwrite*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getaddrinfo − explain getaddrinfo(3) errors

## SYNOPSIS

#include <libexplain/getaddrinfo.h>

const char *explain_errcode_getaddrinfo(int errnum, const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);
void explain_message_errcode_getaddrinfo(char *message, int message_size, int errnum, const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getaddrinfo*(3) system call.

### explain_errcode_getaddrinfo

const char *explain_errcode_getaddrinfo(int errnum, const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);

The **explain_errcode_getaddrinfo** function is used to obtain an explanation of an error returned by the *getaddrinfo*(3) system call. The least the message will contain is the value of `gai_strerror(errcode)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int errcode = getaddrinfo(node, service, hints, res);
if (errncode == GAI_SYSTEM)
    errcode = errno;
if (errcode)
{
    fprintf(stderr, "%s\n", explain_errcode_getaddrinfo(errcode,
        node, service, hints, res));
    exit(EXIT_FAILURE);
}
```

The above code example is available as the *explain_getaddrinfo_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*node* The original node, exactly as passed to the *getaddrinfo*(3) system call.

*service* The original service, exactly as passed to the *getaddrinfo*(3) system call.

*hints* The original hints, exactly as passed to the *getaddrinfo*(3) system call.

*res* The original res, exactly as passed to the *getaddrinfo*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_message_errno_getaddrinfo

void explain_message_errno_getaddrinfo(char *message, int message_size, int errnum, const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);

The **explain_message_errno_getaddrinfo** function may be used to obtain an explanation of an error returned by the *getaddrinfo*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int errcode = getaddrinfo(node, service, hints, res);
if (errnode == EAI_SYSTEM)
    errcode = errno;
if (errcode)
{
    char message[3000];
    explain_message_errcode_getaddrinfo(message, sizeof(message),
        errcode, node, service, hints, res);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getaddrinfo_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*node* The original node, exactly as passed to the *getaddrinfo*(3) system call.

*service* The original service, exactly as passed to the *getaddrinfo*(3) system call.

*hints* The original hints, exactly as passed to the *getaddrinfo*(3) system call.

*res* The original res, exactly as passed to the *getaddrinfo*(3) system call.

## SEE ALSO

*getaddrinfo*(3)
network address and

*explain_getaddrinfo_or_die*(3)
network address and and report errors

## COPYRIGHT

## NAME

explain_getaddrinfo_or_die − network address translation and report errors

## SYNOPSIS

#include <libexplain/getaddrinfo.h>

void explain_getaddrinfo_or_die(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);

## DESCRIPTION

The **explain_getaddrinfo_or_die** function is used to call the *getaddrinfo*(3) system call. On failure, an explanation will be printed to *stderr*, obtained from *explain_getaddrinfo*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_getaddrinfo_or_die(node, service, hints, res);
```

*node* The node, exactly as to be passed to the *getaddrinfo*(3) system call.

*service* The service, exactly as to be passed to the *getaddrinfo*(3) system call.

*hints* The hints, exactly as to be passed to the *getaddrinfo*(3) system call.

*res* The res, exactly as to be passed to the *getaddrinfo*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*getaddrinfo*(3)
network address and service translation

*explain_getaddrinfo*(3)
explain *getaddrinfo*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getc − explain getc(3) errors

## SYNOPSIS

#include <libexplain/getc.h>

const char *explain_getc(FILE *fp);
const char *explain_errno_getc(int errnum, FILE *fp);
void explain_message_getc(char *message, int message_size, FILE *fp);
void explain_message_errno_getc(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getc*(3) system call.

### explain_getc

const char *explain_getc(FILE *fp);

The **explain_getc** function is used to obtain an explanation of an error returned by the *getc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int c = getc(fp);
if (c == EOF && ferror(fp))
{
    fprintf(stderr, "%s\n", explain_getc(fp));
    exit(EXIT_FAILURE);
}
```

*fp* The original fp, exactly as passed to the *getc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_getc

const char *explain_errno_getc(int errnum, FILE *fp);

The **explain_errno_getc** function is used to obtain an explanation of an error returned by the *getc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int c = getc(fp);
if (c == EOF && ferror(fp))
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getc(err, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *getc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_getc**

void explain_message_getc(char *message, int message_size, FILE *fp);

The **explain_message_getc** function may be used to obtain an explanation of an error returned by the *getc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int c = getc(fp);
if (c == EOF && ferror(fp))
{
    char message[3000];
    explain_message_getc(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *getc*(3) system call.

**explain_message_errno_getc**

void explain_message_errno_getc(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_getc** function may be used to obtain an explanation of an error returned by the *getc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int c = getc(fp);
if (c == EOF && ferror(fp))
{
    int err = errno;
    char message[3000];
    explain_message_errno_getc(message, sizeof(message), err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *getc*(3) system call.

## SEE ALSO

*getc*(3) input of characters

*explain_getc_or_die*(3)
input of characters and report errors

## COPYRIGHT

## NAME

explain_getchar − explain getchar(3) errors

## SYNOPSIS

#include <libexplain/getchar.h>

const char *explain_getchar(void);
const char *explain_errno_getchar(int errnum, void);
void explain_message_getchar(char *message, int message_size);
void explain_message_errno_getchar(char *message, int message_size, int errnum);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getchar*(3) system call.

### explain_getchar

const char *explain_getchar(void);

The **explain_getchar** function is used to obtain an explanation of an error returned by the *getchar*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int c = getchar();
if (c == EOF && ferror(stdin))
{
    fprintf(stderr, "%s\n", explain_getchar());
    exit(EXIT_FAILURE);
}
```

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_getchar

const char *explain_errno_getchar(int errnum);

The **explain_errno_getchar** function is used to obtain an explanation of an error returned by the *getchar*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int c = getchar();
if (c == EOF && ferror(stdin))
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getchar(err, ));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**explain_message_getchar**

void explain_message_getchar(char *message, int message_size);

The **explain_message_getchar** function may be used to obtain an explanation of an error returned by the *getchar*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int c = getchar();
if (c == EOF && ferror(stdin))
{
    char message[3000];
    explain_message_getchar(message, sizeof(message), );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

**explain_message_errno_getchar**

void explain_message_errno_getchar(char *message, int message_size, int errnum);

The **explain_message_errno_getchar** function may be used to obtain an explanation of an error returned by the *getchar*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int c = getchar();
if (c == EOF && ferror(stdin))
{
    int err = errno;
    char message[3000];
    explain_message_errno_getchar(message, sizeof(message), err, );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

## SEE ALSO

*getchar*(3)
input of characters

*explain_getchar_or_die*(3)
input of characters and report errors

## COPYRIGHT

## NAME

explain_getchar_or_die − input of characters and report errors

## SYNOPSIS

#include <libexplain/getchar.h>

void explain_getchar_or_die(void);

## DESCRIPTION

The **explain_getchar_or_die** function is used to call the *getchar*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_getchar*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int c = explain_getchar_or_die();
```

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*getchar*(3)
: input of characters

*explain_getchar*(3)
: explain *getchar*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getc_or_die − input of characters and report errors

## SYNOPSIS

#include <libexplain/getc.h>

int explain_getc_or_die(FILE *fp);

## DESCRIPTION

The **explain_getc_or_die** function is used to call the *getc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_getc*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int c = explain_getc_or_die(fp);
```

*fp* The fp, exactly as to be passed to the *getc*(3) system call.

Returns: This function only returns on success, and returns the next character or EOF at end of input. On failure, prints an explanation and exits.

## SEE ALSO

*getc*(3) input of characters

*explain_getc*(3)
explain *getc*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getcwd − explain getcwd(2) errors

## SYNOPSIS

#include <libexplain/getcwd.h>

const char *explain_getcwd(char *buf, size_t size);
const char *explain_errno_getcwd(int errnum, char *buf, size_t size);
void explain_message_getcwd(char *message, int message_size, char *buf, size_t size);
void explain_message_errno_getcwd(char *message, int message_size, int errnum, char *buf, size_t size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getcwd*(2) system call.

### explain_getcwd

const char *explain_getcwd(char *buf, size_t size);

The **explain_getcwd** function is used to obtain an explanation of an error returned by the *getcwd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (getcwd(buf, size) < 0)
{
    fprintf(stderr, "%s\n", explain_getcwd(buf, size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getcwd_or_die*(3) function.

*buf* The original buf, exactly as passed to the *getcwd*(2) system call.

*size* The original size, exactly as passed to the *getcwd*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_getcwd

const char *explain_errno_getcwd(int errnum, char *buf, size_t size);

The **explain_errno_getcwd** function is used to obtain an explanation of an error returned by the *getcwd*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (getcwd(buf, size) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getcwd(err, buf, size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getcwd_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*buf* The original buf, exactly as passed to the *getcwd*(2) system call.

*size* The original size, exactly as passed to the *getcwd*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_getcwd**

void explain_message_getcwd(char *message, int message_size, char *buf, size_t size);

The **explain_message_getcwd** function may be used to obtain an explanation of an error returned by the *getcwd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (getcwd(buf, size) < 0)
{
    char message[3000];
    explain_message_getcwd(message, sizeof(message), buf, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getcwd_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*buf* The original buf, exactly as passed to the *getcwd*(2) system call.

*size* The original size, exactly as passed to the *getcwd*(2) system call.

**explain_message_errno_getcwd**

void explain_message_errno_getcwd(char *message, int message_size, int errnum, char *buf, size_t size);

The **explain_message_errno_getcwd** function may be used to obtain an explanation of an error returned by the *getcwd*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (getcwd(buf, size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_getcwd(message, sizeof(message), err, buf, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getcwd_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*buf*
: The original buf, exactly as passed to the *getcwd*(2) system call.

*size*
: The original size, exactly as passed to the *getcwd*(2) system call.

## SEE ALSO

*getcwd*(2)
: Get current working directory

*explain_getcwd_or_die*(3)
: Get current working directory and report errors

## COPYRIGHT

## NAME

explain_getcwd_or_die − get current working directory and report errors

## SYNOPSIS

#include <libexplain/getcwd.h>

void explain_getcwd_or_die(char *buf, size_t size);

## DESCRIPTION

The **explain_getcwd_or_die** function is used to call the *getcwd*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_getcwd*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_getcwd_or_die(buf, size);
```

*buf* The buf, exactly as to be passed to the *getcwd*(2) system call.

*size* The size, exactly as to be passed to the *getcwd*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*getcwd*(2)
Get current working directory

*explain_getcwd*(3)
explain *getcwd*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getdomainname − explain getdomainname(2) errors

## SYNOPSIS

#include <libexplain/getdomainname.h>

const char *explain_getdomainname(char *data, size_t data_size);
const char *explain_errno_getdomainname(int errnum, char *data, size_t data_size);
void explain_message_getdomainname(char *message, int message_size, char *data, size_t data_size);
void explain_message_errno_getdomainname(char *message, int message_size, int errnum, char *data, size_t data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getdomainname*(2) system call.

### explain_getdomainname

const char *explain_getdomainname(char *data, size_t data_size);

The **explain_getdomainname** function is used to obtain an explanation of an error returned by the *getdomainname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *getdomainname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *getdomainname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getdomainname(data, data_size) < 0)
{
    fprintf(stderr, "%s\n", explain_getdomainname(data,
    data_size));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getdomainname_or_die*(3) function.

### explain_errno_getdomainname

const char *explain_errno_getdomainname(int errnum, char *data, size_t data_size);

The **explain_errno_getdomainname** function is used to obtain an explanation of an error returned by the *getdomainname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *getdomainname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *getdomainname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getdomainname(data, data_size) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_getdomainname(err, data,
   data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getdomainname_or_die*(3) function.

**explain_message_getdomainname**

void explain_message_getdomainname(char *message, int message_size, char *data, size_t data_size);

The **explain_message_getdomainname** function is used to obtain an explanation of an error returned by the *getdomainname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *getdomainname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *getdomainname*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getdomainname(data, data_size) < 0)
{
    char message[3000];
   explain_message_getdomainname(message, sizeof(message), data,
   data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getdomainname_or_die*(3) function.

**explain_message_errno_getdomainname**

void explain_message_errno_getdomainname(char *message, int message_size, int errnum, char *data, size_t data_size);

The **explain_message_errno_getdomainname** function is used to obtain an explanation of an error returned by the *getdomainname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *getdomainname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *getdomainname*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getdomainname(data, data_size) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_getdomainname(message, sizeof(message),
   err, data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getdomainname_or_die*(3) function.

## SEE ALSO

*getdomainname*(2)
get domain name

*explain_getdomainname_or_die*(3)
get domain name and report errors

## COPYRIGHT

## NAME

explain_getdomainname_or_die − get domain name and report errors

## SYNOPSIS

#include <libexplain/getdomainname.h>

void explain_getdomainname_or_die(char *data, size_t data_size);
int explain_getdomainname_on_error(char *data, size_t data_size);

## DESCRIPTION

The **explain_getdomainname_or_die** function is used to call the *getdomainname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getdomainname*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_getdomainname_on_error** function is used to call the *getdomainname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getdomainname*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *getdomainname*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *getdomainname*(2) system call.

## RETURN VALUE

The **explain_getdomainname_or_die** function only returns on success, see *getdomainname*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_getdomainname_on_error** function always returns the value return by the wrapped *getdomainname*(2) system call.

## EXAMPLE

The **explain_getdomainname_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_getdomainname_or_die(data, data_size);
```

## SEE ALSO

*getdomainname*(2)
get domain name

*explain_getdomainname*(3)
explain *getdomainname*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getgroups − explain getgroups(2) errors

## SYNOPSIS

#include <libexplain/getgroups.h>

const char *explain_getgroups(int data_size, gid_t *data);
const char *explain_errno_getgroups(int errnum, int data_size, gid_t *data);
void explain_message_getgroups(char *message, int message_size, int data_size, gid_t *data);
void explain_message_errno_getgroups(char *message, int message_size, int errnum, int data_size, gid_t *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getgroups*(2) system call.

### explain_getgroups

const char *explain_getgroups(int data_size, gid_t *data);

The **explain_getgroups** function is used to obtain an explanation of an error returned by the *getgroups*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data_size*
: The original data_size, exactly as passed to the *getgroups*(2) system call.

*data*
: The original data, exactly as passed to the *getgroups*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getgroups(data_size, data) < 0)
{
   fprintf(stderr, "%s\n", explain_getgroups(data_size, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getgroups_or_die*(3) function.

### explain_errno_getgroups

const char *explain_errno_getgroups(int errnum, int data_size, gid_t *data);

The **explain_errno_getgroups** function is used to obtain an explanation of an error returned by the *getgroups*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data_size*
: The original data_size, exactly as passed to the *getgroups*(2) system call.

*data*
: The original data, exactly as passed to the *getgroups*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getgroups(data_size, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_getgroups(err,
   data_size, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getgroups_or_die*(3) function.

**explain_message_getgroups**

void explain_message_getgroups(char *message, int message_size, int data_size, gid_t *data);

The **explain_message_getgroups** function is used to obtain an explanation of an error returned by the *getgroups*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data_size*
The original data_size, exactly as passed to the *getgroups*(2) system call.

*data* The original data, exactly as passed to the *getgroups*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getgroups(data_size, data) < 0)
{
    char message[3000];
   explain_message_getgroups(message, sizeof(message), data_size,
   data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getgroups_or_die*(3) function.

**explain_message_errno_getgroups**

void explain_message_errno_getgroups(char *message, int message_size, int errnum, int data_size, gid_t *data);

The **explain_message_errno_getgroups** function is used to obtain an explanation of an error returned by the *getgroups*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data_size*
The original data_size, exactly as passed to the *getgroups*(2) system call.

*data* The original data, exactly as passed to the *getgroups*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getgroups(data_size, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_getgroups(message, sizeof(message), err,
   data_size, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getgroups_or_die*(3) function.

## SEE ALSO

*getgroups*(2)
get/set list of supplementary group IDs

*explain_getgroups_or_die*(3)
get/set list of supplementary group IDs and report errors

## COPYRIGHT

## NAME

explain_getgroups_or_die − get supplementary group IDs and report errors

## SYNOPSIS

#include <libexplain/getgroups.h>

void explain_getgroups_or_die(int data_size, gid_t *data);
int explain_getgroups_on_error(int data_size, gid_t *data);

## DESCRIPTION

The **explain_getgroups_or_die** function is used to call the *getgroups*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getgroups*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_getgroups_on_error** function is used to call the *getgroups*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getgroups*(3) function, but still returns to the caller.

*data_size*
: The data_size, exactly as to be passed to the *getgroups*(2) system call.

*data*
: The data, exactly as to be passed to the *getgroups*(2) system call.

## RETURN VALUE

The **explain_getgroups_or_die** function only returns on success, see *getgroups*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_getgroups_on_error** function always returns the value return by the wrapped *getgroups*(2) system call.

## EXAMPLE

The **explain_getgroups_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_getgroups_or_die(data_size, data);
```

## SEE ALSO

*getgroups*(2)
: get/set list of supplementary group IDs

*explain_getgroups*(3)
: explain *getgroups*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_gethostbyname − explain *gethostbyname*(3) errors

## SYNOPSIS

#include <libexplain/gethostbyname.h>

const char *explain_gethostbyname(const char *name);
const char *explain_errno_gethostbyname(int errnum, const char *name);
void explain_message_gethostbyname(char *message, int message_size, const char *name);
void explain_message_errno_gethostbyname(char *message, int message_size, int errnum, const char *name);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *gethostbyname*(3) system call.

### explain_gethostbyname

const char *explain_gethostbyname(const char *name);

The **explain_gethostbyname** function is used to obtain an explanation of an error returned by the *gethostbyname*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*name* The original name, exactly as passed to the *gethostbyname*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
struct hostent *result = gethostbyname(name);
if (!result)
{
   fprintf(stderr, "%s\n", explain_gethostbyname(name));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_gethostbyname_or_die*(3) function.

### explain_errno_gethostbyname

const char *explain_errno_gethostbyname(int errnum, const char *name);

The **explain_errno_gethostbyname** function is used to obtain an explanation of an error returned by the *gethostbyname*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*name* The original name, exactly as passed to the *gethostbyname*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
struct hostent *result = gethostbyname(name);
if (!result)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_gethostbyname(err,
   name));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_gethostbyname_or_die*(3) function.

**explain_message_gethostbyname**

void explain_message_gethostbyname(char *message, int message_size, const char *name);

The **explain_message_gethostbyname** function is used to obtain an explanation of an error returned by the *gethostbyname*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*name* The original name, exactly as passed to the *gethostbyname*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
struct hostent *result = gethostbyname(name);
if (!result)
{
    char message[3000];
   explain_message_gethostbyname(message, sizeof(message), name);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_gethostbyname_or_die*(3) function.

**explain_message_errno_gethostbyname**

void explain_message_errno_gethostbyname(char *message, int message_size, int errnum, const char *name);

The **explain_message_errno_gethostbyname** function is used to obtain an explanation of an error returned by the *gethostbyname*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*name* The original name, exactly as passed to the *gethostbyname*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
struct hostent *result = gethostbyname(name);
if (!result)
```

```
{
    int err = errno;
    char message[3000];
   explain_message_errno_gethostbyname(message, sizeof(message),
   err, name);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_gethostbyname_or_die*(3) function.

## SEE ALSO

*gethostbyname*(3)
get host address given host name

*explain_gethostbyname_or_die*(3)
get host address given host name and report errors

## COPYRIGHT

## NAME

explain_gethostbyname_or_die − get host address by host name and report errors

## SYNOPSIS

#include <libexplain/gethostbyname.h>

struct hostent *explain_gethostbyname_or_die(const char *name);
struct hostent *explain_gethostbyname_on_error(const char *name);

## DESCRIPTION

The **explain_gethostbyname_or_die** function is used to call the *gethostbyname*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_gethostbyname*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_gethostbyname_on_error** function is used to call the *gethostbyname*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_gethostbyname*(3) function, but still returns to the caller.

*name* The name, exactly as to be passed to the *gethostbyname*(3) system call.

## RETURN VALUE

The **explain_gethostbyname_or_die** function only returns on success, see *gethostbyname*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_gethostbyname_on_error** function always returns the value return by the wrapped *gethostbyname*(3) system call.

## EXAMPLE

The **explain_gethostbyname_or_die** function is intended to be used in a fashion similar to the following example:

```
struct hostent *result = explain_gethostbyname_or_die(name);
```

## SEE ALSO

*gethostbyname*(3)
get host address given host name

*explain_gethostbyname*(3)
explain *gethostbyname*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_gethostname − explain gethostname(2) errors

## SYNOPSIS

#include <libexplain/gethostname.h>

const char *explain_gethostname(char *data, size_t data_size);
const char *explain_errno_gethostname(int errnum, char *data, size_t data_size);
void explain_message_gethostname(char *message, int message_size, char *data, size_t data_size);
void explain_message_errno_gethostname(char *message, int message_size, int errnum, char *data, size_t data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *gethostname*(2) system call.

### explain_gethostname

const char *explain_gethostname(char *data, size_t data_size);

The **explain_gethostname** function is used to obtain an explanation of an error returned by the *gethostname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (gethostname(data, data_size) < 0)
{
    fprintf(stderr, "%s\n", explain_gethostname(data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_gethostname_or_die*(3) function.

*data* The original data, exactly as passed to the *gethostname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *gethostname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_gethostname

const char *explain_errno_gethostname(int errnum, char *data, size_t data_size);

The **explain_errno_gethostname** function is used to obtain an explanation of an error returned by the *gethostname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (gethostname(data, data_size) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_gethostname(err, data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_gethostname_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *gethostname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *gethostname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_gethostname**

void explain_message_gethostname(char *message, int message_size, char *data, size_t data_size);

The **explain_message_gethostname** function is used to obtain an explanation of an error returned by the *gethostname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (gethostname(data, data_size) < 0)
{
    char message[3000];
    explain_message_gethostname(message, sizeof(message), data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_gethostname_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *gethostname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *gethostname*(2) system call.

**explain_message_errno_gethostname**

void explain_message_errno_gethostname(char *message, int message_size, int errnum, char *data, size_t data_size);

The **explain_message_errno_gethostname** function is used to obtain an explanation of an error returned by the *gethostname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (gethostname(data, data_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_gethostname(message, sizeof(message), err, data,
        data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_gethostname_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *gethostname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *gethostname*(2) system call.

## SEE ALSO

*gethostname*(2)
get/set hostname

*explain_gethostname_or_die*(3)
get/set hostname and report errors

## COPYRIGHT

## NAME

explain_gethostname_or_die − get/set hostname and report errors

## SYNOPSIS

#include <libexplain/gethostname.h>

void explain_gethostname_or_die(char *data, size_t data_size);
intexplain_gethostname_on_error(char *data, size_t data_size);

## DESCRIPTION

The **explain_gethostname_or_die** function is used to call the *gethostname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_gethostname*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_gethostname_on_error** function is used to call the *gethostname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_gethostname*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *gethostname*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *gethostname*(2) system call.

## RETURN VALUE

The **explain_gethostname_or_die** function only returns on success, see *gethostname*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_gethostname_on_error** function always returns the value return by the wrapped *gethostname*(2) system call.

## EXAMPLE

The **explain_gethostname_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_gethostname_or_die(data, data_size);
```

## SEE ALSO

*gethostname*(2)
get/set hostname

*explain_gethostname*(3)
explain *gethostname*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getpeername − explain getpeername(2) errors

## SYNOPSIS

#include <libexplain/getpeername.h>

const char *explain_getpeername(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);
const char *explain_errno_getpeername(int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);
void explain_message_getpeername(char *message, int message_size, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);
void explain_message_errno_getpeername(char *message, int message_size, int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getpeername*(2) system call.

### explain_getpeername

const char *explain_getpeername(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_getpeername** function is used to obtain an explanation of an error returned by the *getpeername*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
socklen_t sock_addr_len = sizeof(sock_addr);
if (getpeername(fildes, &sock_addr, &sock_addr_size) < 0)
{
    fprintf(stderr, "%s\n", explain_getpeername(fildes,
        &sock_addr, &sock_addr_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getpeername_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *getpeername*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *getpeername*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *getpeername*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_getpeername

const char *explain_errno_getpeername(int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_errno_getpeername** function is used to obtain an explanation of an error returned by the *getpeername*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
```

```
socklen_t sock_addr_len = sizeof(sock_addr);
if (getpeername(fildes, &sock_addr, &sock_addr_size) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getpeername(err,
        fildes, &sock_addr, &sock_addr_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getpeername_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *getpeername*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *getpeername*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *getpeername*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_getpeername**

void explain_message_getpeername(char *message, int message_size, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_message_getpeername** function may be used to obtain an explanation of an error returned by the *getpeername*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
socklen_t sock_addr_len = sizeof(sock_addr);
if (getpeername(fildes, &sock_addr, &sock_addr_size) < 0)
{
    char message[3000];
    explain_message_getpeername(message, sizeof(message),
        fildes, &sock_addr, &sock_addr_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getpeername_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *getpeername*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *getpeername*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *getpeername*(2) system call.

**explain_message_errno_getpeername**

void explain_message_errno_getpeername(char *message, int message_size, int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_message_errno_getpeername** function may be used to obtain an explanation of an error returned by the *getpeername*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
socklen_t sock_addr_len = sizeof(sock_addr);
if (getpeername(fildes, &sock_addr, &sock_addr_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_getpeername(message, sizeof(message),
        err, fildes, &sock_addr, &sock_addr_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getpeername_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *getpeername*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *getpeername*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *getpeername*(2) system call.

## SEE ALSO

*getpeername*(2)
get name of connected peer socket

*explain_getpeername_or_die*(3)
get name of connected peer socket and report errors

## COPYRIGHT

## NAME

explain_getpeername_or_die − get name of peer socket and report errors

## SYNOPSIS

#include <libexplain/getpeername.h>

void explain_getpeername_or_die(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

## DESCRIPTION

The **explain_getpeername_or_die** function is used to call the *getpeername*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_getpeername*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
struct sockadd sock_addr;
socklen_t sock_addr_size = sizeof(sock_addr);
explain_getpeername_or_die(fildes, &sock_addr, &sock_addr_size);
```

*fildes* The fildes, exactly as to be passed to the *getpeername*(2) system call.

*sock_addr*
The sock_addr, exactly as to be passed to the *getpeername*(2) system call.

*sock_addr_size*
The sock_addr_size, exactly as to be passed to the *getpeername*(2) system call.

Returns: This function only returns on success, see *getpeername*(2) for more information. On failure, prints an explanation and exits.

## SEE ALSO

*getpeername*(2)
get name of connected peer socket

*explain_getpeername*(3)
explain *getpeername*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getpgid − explain *getpgid*(2) errors

## SYNOPSIS

#include <libexplain/getpgid.h>

const char *explain_getpgid(pid_t pid);
const char *explain_errno_getpgid(int errnum, pid_t pid);
void explain_message_getpgid(char *message, int message_size, pid_t pid);
void explain_message_errno_getpgid(char *message, int message_size, int errnum, pid_t pid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getpgid*(2) system call.

### explain_getpgid

const char *explain_getpgid(pid_t pid);

The **explain_getpgid** function is used to obtain an explanation of an error returned by the *getpgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pid* The original pid, exactly as passed to the *getpgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = getpgid(pid);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_getpgid(pid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getpgid_or_die*(3) function.

### explain_errno_getpgid

const char *explain_errno_getpgid(int errnum, pid_t pid);

The **explain_errno_getpgid** function is used to obtain an explanation of an error returned by the *getpgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *getpgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = getpgid(pid);
```

```
if (result < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getpgid(err, pid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getpgid_or_die*(3) function.

**explain_message_getpgid**

void explain_message_getpgid(char *message, int message_size, pid_t pid);

The **explain_message_getpgid** function is used to obtain an explanation of an error returned by the *getpgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pid* The original pid, exactly as passed to the *getpgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = getpgid(pid);
if (result < 0)
{
    char message[3000];
    explain_message_getpgid(message, sizeof(message), pid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getpgid_or_die*(3) function.

**explain_message_errno_getpgid**

void explain_message_errno_getpgid(char *message, int message_size, int errnum, pid_t pid);

The **explain_message_errno_getpgid** function is used to obtain an explanation of an error returned by the *getpgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *getpgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = getpgid(pid);
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_getpgid(message, sizeof(message), err,
```

```
                pid);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre−packaged as the *explain_getpgid_or_die*(3) function.

## SEE ALSO

*getpgid*(2)
get process group

*explain_getpgid_or_die*(3)
get process group and report errors

## COPYRIGHT

## NAME

explain_getpgid_or_die − get process group and report errors

## SYNOPSIS

#include <libexplain/getpgid.h>

pid_t explain_getpgid_or_die(pid_t pid);
pid_t explain_getpgid_on_error(pid_t pid);

## DESCRIPTION

The **explain_getpgid_or_die** function is used to call the *getpgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getpgid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_getpgid_on_error** function is used to call the *getpgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getpgid*(3) function, but still returns to the caller.

*pid* The pid, exactly as to be passed to the *getpgid*(2) system call.

## RETURN VALUE

The **explain_getpgid_or_die** function only returns on success, see *getpgid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_getpgid_on_error** function always returns the value return by the wrapped *getpgid*(2) system call.

## EXAMPLE

The **explain_getpgid_or_die** function is intended to be used in a fashion similar to the following example:

```
pid_t result = explain_getpgid_or_die(pid);
```

## SEE ALSO

*getpgid*(2)
get process group

*explain_getpgid*(3)
explain *getpgid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getpgrp − explain *getpgrp*(2) errors

## SYNOPSIS

#include <libexplain/getpgrp.h>

const char *explain_getpgrp(pid_t pid);
const char *explain_errno_getpgrp(int errnum, pid_t pid);
void explain_message_getpgrp(char *message, int message_size, pid_t pid);
void explain_message_errno_getpgrp(char *message, int message_size, int errnum, pid_t pid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getpgrp*(2) system call.

**Note:** the *getpgrp*(2) function has two implementations. The POSIX.1 version has no arguments, while the BSD version has one argument. For simplicity of implementation, the argument list seen here includes the *pid* argument.

The POSIX.1 `getpgid()` semantics can be obtained by calling `getpgrp(0)` on BSD systems, and this is the API for libexplain, even on systems that do not use the BSD API.

### explain_getpgrp

const char *explain_getpgrp(pid_t pid);

The **explain_getpgrp** function is used to obtain an explanation of an error returned by the *getpgrp*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pid* The original pid, exactly as passed to the *getpgrp*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = getpgrp(pid);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_getpgrp(pid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getpgrp_or_die*(3) function.

### explain_errno_getpgrp

const char *explain_errno_getpgrp(int errnum, pid_t pid);

The **explain_errno_getpgrp** function is used to obtain an explanation of an error returned by the *getpgrp*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *getpgrp*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = getpgrp(pid);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_getpgrp(err, pid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getpgrp_or_die*(3) function.

**explain_message_getpgrp**

void explain_message_getpgrp(char *message, int message_size, pid_t pid);

The **explain_message_getpgrp** function is used to obtain an explanation of an error returned by the *getpgrp*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pid* The original pid, exactly as passed to the *getpgrp*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = getpgrp(pid);
if (result < 0)
{
    char message[3000];
   explain_message_getpgrp(message, sizeof(message), pid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getpgrp_or_die*(3) function.

**explain_message_errno_getpgrp**

void explain_message_errno_getpgrp(char *message, int message_size, int errnum, pid_t pid);

The **explain_message_errno_getpgrp** function is used to obtain an explanation of an error returned by the *getpgrp*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *getpgrp*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = getpgrp(pid);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_getpgrp(message, sizeof(message), err,
   pid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getpgrp_or_die*(3) function.

## SEE ALSO

*getpgrp*(2)
get process group

*explain_getpgrp_or_die*(3)
get process group and report errors

## COPYRIGHT

## NAME

explain_getpgrp_or_die − get process group and report errors

## SYNOPSIS

#include <libexplain/getpgrp.h>

pid_t explain_getpgrp_or_die(pid_t pid);
pid_t explain_getpgrp_on_error(pid_t pid);

## DESCRIPTION

The **explain_getpgrp_or_die** function is used to call the *getpgrp*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getpgrp*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_getpgrp_on_error** function is used to call the *getpgrp*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getpgrp*(3) function, but still returns to the caller.

*pid* The pid, exactly as to be passed to the *getpgrp*(2) system call.

### API Inconsistencies

**Note:** the *getpgrp*(2) function has two implementations. The POSIX.1 version has no arguments, while the BSD version has one argument. For simplicity of implementation, the argument list seen here includes the *pid* argument.

The POSIX.1 `getpgid()` semanatics can be obtained by calling `getpgrp(0)` on BSD systems, and this is the API for libexplain, even on systems that do not use the BSD API.

## RETURN VALUE

The **explain_getpgrp_or_die** function only returns on success, see *getpgrp*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_getpgrp_on_error** function always returns the value return by the wrapped *getpgrp*(2) system call.

## EXAMPLE

The **explain_getpgrp_or_die** function is intended to be used in a fashion similar to the following example:

```
pid_t result = explain_getpgrp_or_die(pid);
```

## SEE ALSO

*getpgrp*(2)
get process group

*explain_getpgrp*(3)
explain *getpgrp*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getresgid − explain *getresgid*(2) errors

## SYNOPSIS

#include <libexplain/getresgid.h>

const char *explain_getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
const char *explain_errno_getresgid(int errnum, gid_t *rgid, gid_t *egid, gid_t *sgid);
void explain_message_getresgid(char *message, int message_size, gid_t *rgid, gid_t *egid, gid_t *sgid);
void explain_message_errno_getresgid(char *message, int message_size, int errnum, gid_t *rgid, gid_t *egid, gid_t *sgid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getresgid*(2) system call.

### explain_getresgid

const char *explain_getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);

The **explain_getresgid** function is used to obtain an explanation of an error returned by the *getresgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*rgid* The original rgid, exactly as passed to the *getresgid*(2) system call.

*egid* The original egid, exactly as passed to the *getresgid*(2) system call.

*sgid* The original sgid, exactly as passed to the *getresgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getresgid(rgid, egid, sgid) < 0)
{
   fprintf(stderr, "%s\n", explain_getresgid(rgid, egid, sgid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getresgid_or_die*(3) function.

### explain_errno_getresgid

const char *explain_errno_getresgid(int errnum, gid_t *rgid, gid_t *egid, gid_t *sgid);

The **explain_errno_getresgid** function is used to obtain an explanation of an error returned by the *getresgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*rgid* The original rgid, exactly as passed to the *getresgid*(2) system call.

*egid* The original egid, exactly as passed to the *getresgid*(2) system call.

*sgid* The original sgid, exactly as passed to the *getresgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getresgid(rgid, egid, sgid) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_getresgid(err, rgid,
   egid, sgid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getresgid_or_die*(3) function.

**explain_message_getresgid**

void explain_message_getresgid(char *message, int message_size, gid_t *rgid, gid_t *egid, gid_t *sgid);

The **explain_message_getresgid** function is used to obtain an explanation of an error returned by the *getresgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*rgid* The original rgid, exactly as passed to the *getresgid*(2) system call.

*egid* The original egid, exactly as passed to the *getresgid*(2) system call.

*sgid* The original sgid, exactly as passed to the *getresgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getresgid(rgid, egid, sgid) < 0)
{
    char message[3000];
   explain_message_getresgid(message, sizeof(message), rgid,
   egid, sgid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getresgid_or_die*(3) function.

**explain_message_errno_getresgid**

void explain_message_errno_getresgid(char *message, int message_size, int errnum, gid_t *rgid, gid_t *egid, gid_t *sgid);

The **explain_message_errno_getresgid** function is used to obtain an explanation of an error returned by the *getresgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*rgid* The original rgid, exactly as passed to the *getresgid*(2) system call.

*egid* The original egid, exactly as passed to the *getresgid*(2) system call.

*sgid* The original sgid, exactly as passed to the *getresgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getresgid(rgid, egid, sgid) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_getresgid(message, sizeof(message), err,
   rgid, egid, sgid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getresgid_or_die*(3) function.

## SEE ALSO

*getresgid*(2)
get real, effective and saved group IDs

*explain_getresgid_or_die*(3)
get real, effective and saved group IDs and report errors

## COPYRIGHT

## NAME

explain_getresgid_or_die − get r/e/s group IDs and report errors

## SYNOPSIS

#include <libexplain/getresgid.h>

void explain_getresgid_or_die(gid_t *rgid, gid_t *egid, gid_t *sgid);
int explain_getresgid_on_error(gid_t *rgid, gid_t *egid, gid_t *sgid);

## DESCRIPTION

The **explain_getresgid_or_die** function is used to call the *getresgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getresgid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_getresgid_on_error** function is used to call the *getresgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getresgid*(3) function, but still returns to the caller.

*rgid* The rgid, exactly as to be passed to the *getresgid*(2) system call.

*egid* The egid, exactly as to be passed to the *getresgid*(2) system call.

*sgid* The sgid, exactly as to be passed to the *getresgid*(2) system call.

## RETURN VALUE

The **explain_getresgid_or_die** function only returns on success, see *getresgid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_getresgid_on_error** function always returns the value return by the wrapped *getresgid*(2) system call.

## EXAMPLE

The **explain_getresgid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_getresgid_or_die(rgid, egid, sgid);
```

## SEE ALSO

*getresgid*(2)
get real, effective and saved group IDs

*explain_getresgid*(3)
explain *getresgid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getresuid − explain *getresuid*(2) errors

## SYNOPSIS

#include <libexplain/getresuid.h>

const char *explain_getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
const char *explain_errno_getresuid(int errnum, uid_t *ruid, uid_t *euid, uid_t *suid);
void explain_message_getresuid(char *message, int message_size, uid_t *ruid, uid_t *euid, uid_t *suid);
void explain_message_errno_getresuid(char *message, int message_size, int errnum, uid_t *ruid, uid_t *euid, uid_t *suid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getresuid*(2) system call.

### explain_getresuid

const char *explain_getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);

The **explain_getresuid** function is used to obtain an explanation of an error returned by the *getresuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*ruid* The original ruid, exactly as passed to the *getresuid*(2) system call.

*euid* The original euid, exactly as passed to the *getresuid*(2) system call.

*suid* The original suid, exactly as passed to the *getresuid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getresuid(ruid, euid, suid) < 0)
{
   fprintf(stderr, "%s\n", explain_getresuid(ruid, euid, suid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getresuid_or_die*(3) function.

### explain_errno_getresuid

const char *explain_errno_getresuid(int errnum, uid_t *ruid, uid_t *euid, uid_t *suid);

The **explain_errno_getresuid** function is used to obtain an explanation of an error returned by the *getresuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ruid* The original ruid, exactly as passed to the *getresuid*(2) system call.

*euid* The original euid, exactly as passed to the *getresuid*(2) system call.

*suid* The original suid, exactly as passed to the *getresuid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getresuid(ruid, euid, suid) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_getresuid(err, ruid,
   euid, suid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getresuid_or_die*(3) function.

**explain_message_getresuid**

void explain_message_getresuid(char *message, int message_size, uid_t *ruid, uid_t *euid, uid_t *suid);

The **explain_message_getresuid** function is used to obtain an explanation of an error returned by the *getresuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*ruid* The original ruid, exactly as passed to the *getresuid*(2) system call.

*euid* The original euid, exactly as passed to the *getresuid*(2) system call.

*suid* The original suid, exactly as passed to the *getresuid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getresuid(ruid, euid, suid) < 0)
{
    char message[3000];
   explain_message_getresuid(message, sizeof(message), ruid,
   euid, suid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getresuid_or_die*(3) function.

**explain_message_errno_getresuid**

void explain_message_errno_getresuid(char *message, int message_size, int errnum, uid_t *ruid, uid_t *euid, uid_t *suid);

The **explain_message_errno_getresuid** function is used to obtain an explanation of an error returned by the *getresuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ruid* The original ruid, exactly as passed to the *getresuid*(2) system call.

*euid* The original euid, exactly as passed to the *getresuid*(2) system call.

*suid* The original suid, exactly as passed to the *getresuid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getresuid(ruid, euid, suid) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_getresuid(message, sizeof(message), err,
   ruid, euid, suid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getresuid_or_die*(3) function.

## SEE ALSO

*getresuid*(2)
get real, effective and saved user IDs

*explain_getresuid_or_die*(3)
get real, effective and saved user IDs and report errors

## COPYRIGHT

## NAME

explain_getresuid_or_die − get r/e/s user IDs and report errors

## SYNOPSIS

#include <libexplain/getresuid.h>

void explain_getresuid_or_die(uid_t *ruid, uid_t *euid, uid_t *suid);
int explain_getresuid_on_error(uid_t *ruid, uid_t *euid, uid_t *suid);

## DESCRIPTION

The **explain_getresuid_or_die** function is used to call the *getresuid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getresuid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_getresuid_on_error** function is used to call the *getresuid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getresuid*(3) function, but still returns to the caller.

*ruid* The ruid, exactly as to be passed to the *getresuid*(2) system call.

*euid* The euid, exactly as to be passed to the *getresuid*(2) system call.

*suid* The suid, exactly as to be passed to the *getresuid*(2) system call.

## RETURN VALUE

The **explain_getresuid_or_die** function only returns on success, see *getresuid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_getresuid_on_error** function always returns the value return by the wrapped *getresuid*(2) system call.

## EXAMPLE

The **explain_getresuid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_getresuid_or_die(ruid, euid, suid);
```

## SEE ALSO

*getresuid*(2)
get real, effective and saved user IDs

*explain_getresuid*(3)
explain *getresuid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getrlimit − explain getrlimit(2) errors

## SYNOPSIS

#include <libexplain/getrlimit.h>

const char *explain_getrlimit(int resource, struct rlimit *rlim);
const char *explain_errno_getrlimit(int errnum, int resource, struct rlimit *rlim);
void explain_message_getrlimit(char *message, int message_size, int resource, struct rlimit *rlim);
void explain_message_errno_getrlimit(char *message, int message_size, int errnum, int resource, struct rlimit *rlim);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getrlimit*(2) system call.

### explain_getrlimit

const char *explain_getrlimit(int resource, struct rlimit *rlim);

The **explain_getrlimit** function is used to obtain an explanation of an error returned by the *getrlimit*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (getrlimit(resource, rlim) < 0)
{
    fprintf(stderr, "%s\n", explain_getrlimit(resource, rlim));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getrlimit_or_die*(3) function.

*resource* The original resource, exactly as passed to the *getrlimit*(2) system call.

*rlim* The original rlim, exactly as passed to the *getrlimit*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_getrlimit

const char *explain_errno_getrlimit(int errnum, int resource, struct rlimit *rlim);

The **explain_errno_getrlimit** function is used to obtain an explanation of an error returned by the *getrlimit*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (getrlimit(resource, rlim) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getrlimit(err, resource, rlim));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getrlimit_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*resource* The original resource, exactly as passed to the *getrlimit*(2) system call.

*rlim* The original rlim, exactly as passed to the *getrlimit*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_getrlimit**

void explain_message_getrlimit(char *message, int message_size, int resource, struct rlimit *rlim);

The **explain_message_getrlimit** function may be used to obtain an explanation of an error returned by the *getrlimit*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (getrlimit(resource, rlim) < 0)
{
    char message[3000];
    explain_message_getrlimit(message, sizeof(message), resource, rlim);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getrlimit_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*resource* The original resource, exactly as passed to the *getrlimit*(2) system call.

*rlim* The original rlim, exactly as passed to the *getrlimit*(2) system call.

**explain_message_errno_getrlimit**

void explain_message_errno_getrlimit(char *message, int message_size, int errnum, int resource, struct rlimit *rlim);

The **explain_message_errno_getrlimit** function may be used to obtain an explanation of an error returned by the *getrlimit*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (getrlimit(resource, rlim) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_getrlimit(message, sizeof(message),
        err, resource, rlim);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getrlimit_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*resource* The original resource, exactly as passed to the *getrlimit*(2) system call.

*rlim* The original rlim, exactly as passed to the *getrlimit*(2) system call.

## SEE ALSO

*getrlimit*(2)
get resource limits

*explain_getrlimit_or_die*(3)
get resource limits and report errors

## COPYRIGHT

## NAME

explain_getrlimit_or_die − get resource limits and report errors

## SYNOPSIS

#include <libexplain/getrlimit.h>

void explain_getrlimit_or_die(int resource, struct rlimit *rlim);

## DESCRIPTION

The **explain_getrlimit_or_die** function is used to call the *getrlimit*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_getrlimit*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_getrlimit_or_die(resource, rlim);
```

*resource* The resource, exactly as to be passed to the *getrlimit*(2) system call.

*rlim* The rlim, exactly as to be passed to the *getrlimit*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*getrlimit*(2)
get resource limits

*explain_getrlimit*(3)
explain *getrlimit*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getrusage − explain *getrusage*(2) errors

## SYNOPSIS

#include <libexplain/getrusage.h>

const char *explain_getrusage(int who, struct rusage *usage);
const char *explain_errno_getrusage(int errnum, int who, struct rusage *usage);
void explain_message_getrusage(char *message, int message_size, int who, struct rusage *usage);
void explain_message_errno_getrusage(char *message, int message_size, int errnum, int who, struct rusage *usage);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getrusage*(2) system call.

### explain_getrusage

const char *explain_getrusage(int who, struct rusage *usage);

The **explain_getrusage** function is used to obtain an explanation of an error returned by the *getrusage*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*who* The original who, exactly as passed to the *getrusage*(2) system call.

*usage* The original usage, exactly as passed to the *getrusage*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getrusage(who, usage) < 0)
{
    fprintf(stderr, "%s\n", explain_getrusage(who, usage));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getrusage_or_die*(3) function.

### explain_errno_getrusage

const char *explain_errno_getrusage(int errnum, int who, struct rusage *usage);

The **explain_errno_getrusage** function is used to obtain an explanation of an error returned by the *getrusage*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*who* The original who, exactly as passed to the *getrusage*(2) system call.

*usage* The original usage, exactly as passed to the *getrusage*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getrusage(who, usage) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getrusage(err, who,
    usage));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getrusage_or_die*(3) function.

**explain_message_getrusage**

void explain_message_getrusage(char *message, int message_size, int who, struct rusage *usage);

The **explain_message_getrusage** function is used to obtain an explanation of an error returned by the *getrusage*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*who* The original who, exactly as passed to the *getrusage*(2) system call.

*usage* The original usage, exactly as passed to the *getrusage*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getrusage(who, usage) < 0)
{
    char message[3000];
    explain_message_getrusage(message, sizeof(message), who,
    usage);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getrusage_or_die*(3) function.

**explain_message_errno_getrusage**

void explain_message_errno_getrusage(char *message, int message_size, int errnum, int who, struct rusage *usage);

The **explain_message_errno_getrusage** function is used to obtain an explanation of an error returned by the *getrusage*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*who* The original who, exactly as passed to the *getrusage*(2) system call.

*usage* The original usage, exactly as passed to the *getrusage*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (getrusage(who, usage) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_getrusage(message, sizeof(message), err,
   who, usage);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_getrusage_or_die*(3) function.

## SEE ALSO

*getrusage*(2)
get resource usage

*explain_getrusage_or_die*(3)
get resource usage and report errors

## COPYRIGHT

## NAME

explain_getrusage_or_die − get resource usage and report errors

## SYNOPSIS

#include <libexplain/getrusage.h>

void explain_getrusage_or_die(int who, struct rusage *usage);
int explain_getrusage_on_error(int who, struct rusage *usage);

## DESCRIPTION

The **explain_getrusage_or_die** function is used to call the *getrusage*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getrusage*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_getrusage_on_error** function is used to call the *getrusage*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getrusage*(3) function, but still returns to the caller.

*who* The who, exactly as to be passed to the *getrusage*(2) system call.

*usage* The usage, exactly as to be passed to the *getrusage*(2) system call.

## RETURN VALUE

The **explain_getrusage_or_die** function only returns on success, see *getrusage*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_getrusage_on_error** function always returns the value return by the wrapped *getrusage*(2) system call.

## EXAMPLE

The **explain_getrusage_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_getrusage_or_die(who, usage);
```

## SEE ALSO

*getrusage*(2)
get resource usage

*explain_getrusage*(3)
explain *getrusage*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getsockname − explain getsockname(2) errors

## SYNOPSIS

#include <libexplain/getsockname.h>

const char *explain_getsockname(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);
const char *explain_errno_getsockname(int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);
void explain_message_getsockname(char *message, int message_size, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);
void explain_message_errno_getsockname(char *message, int message_size, int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getsockname*(2) system call.

### explain_getsockname

const char *explain_getsockname(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_getsockname** function is used to obtain an explanation of an error returned by the *getsockname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
socklen_t sock_addr_size = sizeof(sock_addr);
if (getsockname(fildes, &sock_addr, &sock_addr_size) < 0)
{
    fprintf(stderr, "%s\n", explain_getsockname(fildes,
        &sock_addr, &sock_addr_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getsockname_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *getsockname*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *getsockname*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *getsockname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_getsockname

const char *explain_errno_getsockname(int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_errno_getsockname** function is used to obtain an explanation of an error returned by the *getsockname*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
```

```
socklen_t sock_addr_size = sizeof(sock_addr);
if (getsockname(fildes, &sock_addr, &sock_addr_size) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getsockname(err,
        fildes, &sock_addr, &sock_addr_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getsockname_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *getsockname*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *getsockname*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *getsockname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_getsockname**

void explain_message_getsockname(char *message, int message_size, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_message_getsockname** function may be used to obtain an explanation of an error returned by the *getsockname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
socklen_t sock_addr_size = sizeof(sock_addr);
if (getsockname(fildes, &sock_addr, &sock_addr_size) < 0)
{
    char message[3000];
    explain_message_getsockname(message, sizeof(message),
        fildes, &sock_addr, &sock_addr_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getsockname_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *getsockname*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *getsockname*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *getsockname*(2) system call.

**explain_message_errno_getsockname**
void explain_message_errno_getsockname(char *message, int message_size, int errnum, int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

The **explain_message_errno_getsockname** function may be used to obtain an explanation of an error returned by the *getsockname*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
socklen_t sock_addr_size = sizeof(sock_addr);
if (getsockname(fildes, &sock_addr, &sock_addr_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_getsockname(message, sizeof(message),
        err, fildes, &sock_addr, &sock_addr_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getsockname_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *getsockname*(2) system call.

*sock_addr*
The original sock_addr, exactly as passed to the *getsockname*(2) system call.

*sock_addr_size*
The original sock_addr_size, exactly as passed to the *getsockname*(2) system call.

## SEE ALSO

*getsockname*(2)
get socket name

*explain_getsockname_or_die*(3)
get socket name and report errors

## COPYRIGHT

## NAME

explain_getsockname_or_die − get socket name and report errors

## SYNOPSIS

#include <libexplain/getsockname.h>

void explain_getsockname_or_die(int fildes, struct sockaddr *sock_addr, socklen_t *sock_addr_size);

## DESCRIPTION

The **explain_getsockname_or_die** function is used to call the *getsockname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_getsockname*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
struct sockaddr sock_addr;
socklen_t sock_addr_size = sizeof(sock_addr);
explain_getsockname_or_die(fildes, &sock_addr, &sock_addr_size);
```

*fildes* The fildes, exactly as to be passed to the *getsockname*(2) system call.

*sock_addr*
The sock_addr, exactly as to be passed to the *getsockname*(2) system call.

*sock_addr_size*
The sock_addr_size, exactly as to be passed to the *getsockname*(2) system call.

Returns: This function only returns on success, see *getsockaddr*(1) for more information. On failure, prints an explanation and exits.

## SEE ALSO

*getsockname*(2)
get socket name

*explain_getsockname*(3)
explain *getsockname*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getsockopt − explain getsockopt(2) errors

## SYNOPSIS

#include <libexplain/getsockopt.h>

const char *explain_getsockopt(int fildes, int level, int name, void *data, socklen_t *data_size);
const char *explain_errno_getsockopt(int errnum, int fildes, int level, int name, void *data, socklen_t *data_size);
void explain_message_getsockopt(char *message, int message_size, int fildes, int level, int name, void *data, socklen_t *data_size);
void explain_message_errno_getsockopt(char *message, int message_size, int errnum, int fildes, int level, int name, void *data, socklen_t *data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getsockopt*(2) system call.

### explain_getsockopt

const char *explain_getsockopt(int fildes, int level, int name, void *data, socklen_t *data_size);

The **explain_getsockopt** function is used to obtain an explanation of an error returned by the *getsockopt*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (getsockopt(fildes, level, name, data, data_size) < 0)
{
    fprintf(stderr, "%s\n", explain_getsockopt(fildes,
        level, name, data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getsockopt_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *getsockopt*(2) system call.

*level* The original level, exactly as passed to the *getsockopt*(2) system call.

*name* The original name, exactly as passed to the *getsockopt*(2) system call.

*data* The original data, exactly as passed to the *getsockopt*(2) system call.

*data_size*
The original data_size, exactly as passed to the *getsockopt*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_getsockopt

const char *explain_errno_getsockopt(int errnum, int fildes, int level, int name, void *data, socklen_t *data_size);

The **explain_errno_getsockopt** function is used to obtain an explanation of an error returned by the *getsockopt*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (getsockopt(fildes, level, name, data, data_size) < 0)
```

```
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getsockopt(err,
        fildes, level, name, data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getsockopt_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *getsockopt*(2) system call.

*level* The original level, exactly as passed to the *getsockopt*(2) system call.

*name* The original name, exactly as passed to the *getsockopt*(2) system call.

*data* The original data, exactly as passed to the *getsockopt*(2) system call.

*data_size*
The original data_size, exactly as passed to the *getsockopt*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_getsockopt**

void explain_message_getsockopt(char *message, int message_size, int fildes, int level, int name, void *data, socklen_t *data_size);

The **explain_message_getsockopt** function may be used to obtain an explanation of an error returned by the *getsockopt*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (getsockopt(fildes, level, name, data, data_size) < 0)
{
    char message[3000];
    explain_message_getsockopt(message, sizeof(message),
        fildes, level, name, data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getsockopt_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *getsockopt*(2) system call.

*level* The original level, exactly as passed to the *getsockopt*(2) system call.

*name* The original name, exactly as passed to the *getsockopt*(2) system call.

*data* The original data, exactly as passed to the *getsockopt*(2) system call.

*data_size*
The original data_size, exactly as passed to the *getsockopt*(2) system call.

**explain_message_errno_getsockopt**

void explain_message_errno_getsockopt(char *message, int message_size, int errnum, int fildes, int level, int name, void *data, socklen_t *data_size);

The **explain_message_errno_getsockopt** function may be used to obtain an explanation of an error returned by the *getsockopt*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (getsockopt(fildes, level, name, data, data_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_getsockopt(message, sizeof(message),
        err, fildes, level, name, data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getsockopt_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *getsockopt*(2) system call.

*level* The original level, exactly as passed to the *getsockopt*(2) system call.

*name* The original name, exactly as passed to the *getsockopt*(2) system call.

*data* The original data, exactly as passed to the *getsockopt*(2) system call.

*data_size*
The original data_size, exactly as passed to the *getsockopt*(2) system call.

## SEE ALSO

*getsockopt*(2)
get and set options on sockets

*explain_getsockopt_or_die*(3)
get and set options on sockets and report errors

## COPYRIGHT

## NAME

explain_getsockopt_or_die − get and set options on sockets and report errors

## SYNOPSIS

#include <libexplain/getsockopt.h>

void explain_getsockopt_or_die(int fildes, int level, int name, void *data, socklen_t *data_size);

## DESCRIPTION

The **explain_getsockopt_or_die** function is used to call the *getsockopt*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_getsockopt*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_getsockopt_or_die(fildes, level, name, data, data_size);
```

*fildes* The fildes, exactly as to be passed to the *getsockopt*(2) system call.

*level* The level, exactly as to be passed to the *getsockopt*(2) system call.

*name* The name, exactly as to be passed to the *getsockopt*(2) system call.

*data* The data, exactly as to be passed to the *getsockopt*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *getsockopt*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*getsockopt*(2)
get and set options on sockets

*explain_getsockopt*(3)
explain *getsockopt*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_gettimeofday − explain gettimeofday(2) errors

## SYNOPSIS

#include <libexplain/gettimeofday.h>

const char *explain_gettimeofday(struct timeval *tv, struct timezone *tz);
const char *explain_errno_gettimeofday(int errnum, struct timeval *tv, struct timezone *tz);
void explain_message_gettimeofday(char *message, int message_size, struct timeval *tv, struct timezone *tz);
void explain_message_errno_gettimeofday(char *message, int message_size, int errnum, struct timeval *tv, struct timezone *tz);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *gettimeofday*(2) system call.

### explain_gettimeofday

const char *explain_gettimeofday(struct timeval *tv, struct timezone *tz);

The **explain_gettimeofday** function is used to obtain an explanation of an error returned by the *gettimeofday*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (gettimeofday(tv, tz) < 0)
{
    fprintf(stderr, "%s\n", explain_gettimeofday(tv, tz));
    exit(EXIT_FAILURE);
}
```

*tv* The original tv, exactly as passed to the *gettimeofday*(2) system call.

*tz* The original tz, exactly as passed to the *gettimeofday*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_gettimeofday

const char *explain_errno_gettimeofday(int errnum, struct timeval *tv, struct timezone *tz);

The **explain_errno_gettimeofday** function is used to obtain an explanation of an error returned by the *gettimeofday*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (gettimeofday(tv, tz) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_gettimeofday(err, tv, tz));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*tv* The original tv, exactly as passed to the *gettimeofday*(2) system call.

*tz* The original tz, exactly as passed to the *gettimeofday*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_gettimeofday**

void explain_message_gettimeofday(char *message, int message_size, struct timeval *tv, struct timezone *tz);

The **explain_message_gettimeofday** function may be used to obtain an explanation of an error returned by the *gettimeofday*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (gettimeofday(tv, tz) < 0)
{
    char message[3000];
    explain_message_gettimeofday(message, sizeof(message), tv, tz);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*tv* The original tv, exactly as passed to the *gettimeofday*(2) system call.

*tz* The original tz, exactly as passed to the *gettimeofday*(2) system call.

**explain_message_errno_gettimeofday**

void explain_message_errno_gettimeofday(char *message, int message_size, int errnum, struct timeval *tv, struct timezone *tz);

The **explain_message_errno_gettimeofday** function may be used to obtain an explanation of an error returned by the *gettimeofday*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (gettimeofday(tv, tz) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_gettimeofday(message, sizeof(message), err,
        tv, tz);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*tv* The original tv, exactly as passed to the *gettimeofday*(2) system call.

*tz* The original tz, exactly as passed to the *gettimeofday*(2) system call.

## SEE ALSO

*gettimeofday*(2)
get time

*explain_gettimeofday_or_die*(3)
get time and report errors

## COPYRIGHT

## NAME

explain_gettimeofday_or_die − get time and report errors

## SYNOPSIS

#include <libexplain/gettimeofday.h>

void explain_gettimeofday_or_die(struct timeval *tv, struct timezone *tz);

## DESCRIPTION

The **explain_gettimeofday_or_die** function is used to call the *gettimeofday*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_gettimeofday*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_gettimeofday_or_die(tv, tz);
```

*tv* The tv, exactly as to be passed to the *gettimeofday*(2) system call.

*tz* The tz, exactly as to be passed to the *gettimeofday*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*gettimeofday*(2)
get time

*explain_gettimeofday*(3)
explain *gettimeofday*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_getw − explain *getw*(3) errors

## SYNOPSIS

#include <libexplain/getw.h>

const char *explain_getw(FILE *fp);
const char *explain_errno_getw(int errnum, FILE *fp);
void explain_message_getw(char *message, int message_size, FILE *fp);
void explain_message_errno_getw(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *getw*(3) system call.

### explain_getw

const char *explain_getw(FILE *fp);

The **explain_getw** function is used to obtain an explanation of an error returned by the *getw*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *getw*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = getw(fp);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_getw(fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getw_or_die*(3) function.

### explain_errno_getw

const char *explain_errno_getw(int errnum, FILE *fp);

The **explain_errno_getw** function is used to obtain an explanation of an error returned by the *getw*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *getw*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = getw(fp);
```

```
if (result < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_getw(err, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getw_or_die*(3) function.

**explain_message_getw**

void explain_message_getw(char *message, int message_size, FILE *fp);

The **explain_message_getw** function is used to obtain an explanation of an error returned by the *getw*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *getw*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = getw(fp);
if (result < 0)
{
    char message[3000];
    explain_message_getw(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_getw_or_die*(3) function.

**explain_message_errno_getw**

void explain_message_errno_getw(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_getw** function is used to obtain an explanation of an error returned by the *getw*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *getw*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = getw(fp);
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_getw(message, sizeof(message), err, fp);
```

```
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_getw_or_die*(3) function.

## SEE ALSO

*getw*(3) input a word (int)

*explain_getw_or_die*(3)
input a word (int) and report errors

## COPYRIGHT

## NAME

explain_getw_or_die − input a word (int) and report errors

## SYNOPSIS

#include <libexplain/getw.h>

int explain_getw_or_die(FILE *fp);
int explain_getw_on_error(FILE *fp);

## DESCRIPTION

The **explain_getw_or_die** function is used to call the *getw*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getw*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_getw_on_error** function is used to call the *getw*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_getw*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *getw*(3) system call.

## RETURN VALUE

The **explain_getw_or_die** function only returns on success, see *getw*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_getw_on_error** function always returns the value return by the wrapped *getw*(3) system call.

## EXAMPLE

The **explain_getw_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_getw_or_die(fp);
```

## SEE ALSO

*getw*(3) input a word (int)

*explain_getw*(3)
explain *getw*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_ioctl − explain ioctl(2) errors

## SYNOPSIS

#include <libexplain/ioctl.h>

const char *explain_ioctl(int fildes, int request, void *data);
const char *explain_errno_ioctl(int errnum, int fildes, int request, void *data);
void explain_message_ioctl(char *message, int message_size, int fildes, int request, void *data);
void explain_message_errno_ioctl(char *message, int message_size, int errnum, int fildes, int request, void *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *ioctl*(2) system call.

### explain_ioctl

const char *explain_ioctl(int fildes, int request, void *data);

The **explain_ioctl** function is used to obtain an explanation of an error returned by the *ioctl*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int result = ioctl(fildes, request, data);
if (result < 0)
{
    fprintf(stderr, "%s\n", explain_ioctl(fildes, request, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ioctl_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *ioctl*(2) system call.

*request* The original request, exactly as passed to the *ioctl*(2) system call.

*data* The original data, exactly as passed to the *ioctl*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_ioctl

const char *explain_errno_ioctl(int errnum, int fildes, int request, void *data);

The **explain_errno_ioctl** function is used to obtain an explanation of an error returned by the *ioctl*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (ioctl(fildes, request, data) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n",
        explain_errno_ioctl(err, fildes, request, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ioctl_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *ioctl*(2) system call.

*request* The original request, exactly as passed to the *ioctl*(2) system call.

*data* The original data, exactly as passed to the *ioctl*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_ioctl**

void explain_message_ioctl(char *message, int message_size, int fildes, int request, void *data);

The **explain_message_ioctl** function may be used to obtain an explanation of an error returned by the *ioctl*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (ioctl(fildes, request, data) < 0)
{
    char message[3000];
    explain_message_ioctl(message, sizeof(message), fildes, request, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ioctl_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *ioctl*(2) system call.

*request* The original request, exactly as passed to the *ioctl*(2) system call.

*data* The original data, exactly as passed to the *ioctl*(2) system call.

**explain_message_errno_ioctl**

void explain_message_errno_ioctl(char *message, int message_size, int errnum, int fildes, int request, void *data);

The **explain_message_errno_ioctl** function may be used to obtain an explanation of an error returned by the *ioctl*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (ioctl(fildes, request, data) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_ioctl(message, sizeof(message), err,
        fildes, request, data);
    fprintf(stderr, "%s\n", message);
```

```
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_ioctl_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *ioctl*(2) system call.

*request* The original request, exactly as passed to the *ioctl*(2) system call.

*data* The original data, exactly as passed to the *ioctl*(2) system call.

## SEE ALSO

*ioctl*(2) control device

*explain_ioctl_or_die*(3)
control device and report errors

## COPYRIGHT

## NAME

explain_ioctl_or_die − control device and report errors

## SYNOPSIS

#include <libexplain/ioctl.h>

int explain_ioctl_or_die(int fildes, int request, void *data);

## DESCRIPTION

The **explain_ioctl_or_die** function is used to call the *ioctl*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_ioctl*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int result = explain_ioctl_or_die(fildes, request, data);
```

*fildes* The fildes, exactly as to be passed to the *ioctl*(2) system call.

*request* The request, exactly as to be passed to the *ioctl*(2) system call.

*data* The data, exactly as to be passed to the *ioctl*(2) system call.

Returns: This function only returns on success, see *ioctl*(2) for more information. On failure, prints an explanation and exit()s.

## SEE ALSO

*ioctl*(2) control device

*explain_ioctl*(3)
explain *ioctl*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_kill − explain kill(2) errors

## SYNOPSIS

#include <libexplain/kill.h>

const char *explain_kill(pid_t pid, int sig);
const char *explain_errno_kill(int errnum, pid_t pid, int sig);
void explain_message_kill(char *message, int message_size, pid_t pid, int sig);
void explain_message_errno_kill(char *message, int message_size, int errnum, pid_t pid, int sig);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *kill*(2) system call.

### explain_kill

const char *explain_kill(pid_t pid, int sig);

The **explain_kill** function is used to obtain an explanation of an error returned by the *kill*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pid* The original pid, exactly as passed to the *kill*(2) system call.

*sig* The original sig, exactly as passed to the *kill*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (kill(pid, sig) < 0)
{
   fprintf(stderr, "%s\n", explain_kill(pid, sig));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_kill_or_die*(3) function.

### explain_errno_kill

const char *explain_errno_kill(int errnum, pid_t pid, int sig);

The **explain_errno_kill** function is used to obtain an explanation of an error returned by the *kill*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *kill*(2) system call.

*sig* The original sig, exactly as passed to the *kill*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (kill(pid, sig) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_kill(err, pid, sig));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_kill_or_die*(3) function.

**explain_message_kill**

void explain_message_kill(char *message, int message_size, pid_t pid, int sig);

The **explain_message_kill** function is used to obtain an explanation of an error returned by the *kill*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pid* The original pid, exactly as passed to the *kill*(2) system call.

*sig* The original sig, exactly as passed to the *kill*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (kill(pid, sig) < 0)
{
    char message[3000];
    explain_message_kill(message, sizeof(message), pid, sig);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_kill_or_die*(3) function.

**explain_message_errno_kill**

void explain_message_errno_kill(char *message, int message_size, int errnum, pid_t pid, int sig);

The **explain_message_errno_kill** function is used to obtain an explanation of an error returned by the *kill*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *kill*(2) system call.

*sig* The original sig, exactly as passed to the *kill*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (kill(pid, sig) < 0)
{
    int err = errno;
    char message[3000];
```

```
        explain_message_errno_kill(message, sizeof(message), err, pid,
        sig);
         fprintf(stderr, "%s\n", message);
         exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_kill_or_die*(3) function.

## SEE ALSO

*kill*(2) send signal to a process

*explain_kill_or_die*(3)
send signal to a process and report errors

## COPYRIGHT

## NAME

explain_kill_or_die − send signal to a process and report errors

## SYNOPSIS

#include <libexplain/kill.h>

void explain_kill_or_die(pid_t pid, int sig);
int explain_kill_on_error(pid_t pid, int sig);

## DESCRIPTION

The **explain_kill_or_die** function is used to call the *kill*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_kill*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_kill_on_error** function is used to call the *kill*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_kill*(3) function, but still returns to the caller.

*pid* The pid, exactly as to be passed to the *kill*(2) system call.

*sig* The sig, exactly as to be passed to the *kill*(2) system call.

## RETURN VALUE

The **explain_kill_or_die** function only returns on success, see *kill*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_kill_on_error** function always returns the value return by the wrapped *kill*(2) system call.

## EXAMPLE

The **explain_kill_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_kill_or_die(pid, sig);
```

## SEE ALSO

*kill*(2) send signal to a process

*explain_kill*(3)
explain *kill*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_lchmod − explain *lchmod*(2) errors

## SYNOPSIS

#include <libexplain/lchmod.h>

const char *explain_lchmod(const char *pathname, mode_t mode);
const char *explain_errno_lchmod(int errnum, const char *pathname, mode_t mode);
void explain_message_lchmod(char *message, int message_size, const char *pathname, mode_t mode);
void explain_message_errno_lchmod(char *message, int message_size, int errnum, const char *pathname, mode_t mode);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *lchmod*(2) system call.

### explain_lchmod

const char *explain_lchmod(const char *pathname, mode_t mode);

The **explain_lchmod** function is used to obtain an explanation of an error returned by the *lchmod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *lchmod*(2) system call.

*mode*
: The original mode, exactly as passed to the *lchmod*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (lchmod(pathname, mode) < 0)
{
   fprintf(stderr, "%s\n", explain_lchmod(pathname, mode));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_lchmod_or_die*(3) function.

### explain_errno_lchmod

const char *explain_errno_lchmod(int errnum, const char *pathname, mode_t mode);

The **explain_errno_lchmod** function is used to obtain an explanation of an error returned by the *lchmod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *lchmod*(2) system call.

*mode*
: The original mode, exactly as passed to the *lchmod*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (lchmod(pathname, mode) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_lchmod(err, pathname,
   mode));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_lchmod_or_die*(3) function.

**explain_message_lchmod**

void explain_message_lchmod(char *message, int message_size, const char *pathname, mode_t mode);

The **explain_message_lchmod** function is used to obtain an explanation of an error returned by the *lchmod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *lchmod*(2) system call.

*mode* The original mode, exactly as passed to the *lchmod*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (lchmod(pathname, mode) < 0)
{
    char message[3000];
   explain_message_lchmod(message, sizeof(message), pathname,
   mode);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_lchmod_or_die*(3) function.

**explain_message_errno_lchmod**

void explain_message_errno_lchmod(char *message, int message_size, int errnum, const char *pathname, mode_t mode);

The **explain_message_errno_lchmod** function is used to obtain an explanation of an error returned by the *lchmod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *lchmod*(2) system call.

*mode* The original mode, exactly as passed to the *lchmod*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (lchmod(pathname, mode) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_lchmod(message, sizeof(message), err,
   pathname, mode);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_lchmod_or_die*(3) function.

## SEE ALSO

*lchmod*(2)
change permissions of a file

*explain_lchmod_or_die*(3)
change permissions of a file and report errors

## COPYRIGHT

## NAME

explain_lchmod_or_die − change permissions of a file and report errors

## SYNOPSIS

#include <libexplain/lchmod.h>

void explain_lchmod_or_die(const char *pathname, mode_t mode);
int explain_lchmod_on_error(const char *pathname, mode_t mode);

## DESCRIPTION

The **explain_lchmod_or_die** function is used to call the *lchmod*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_lchmod*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_lchmod_on_error** function is used to call the *lchmod*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_lchmod*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *lchmod*(2) system call.

*mode*
: The mode, exactly as to be passed to the *lchmod*(2) system call.

## RETURN VALUE

The **explain_lchmod_or_die** function only returns on success, see *lchmod*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_lchmod_on_error** function always returns the value return by the wrapped *lchmod*(2) system call.

## EXAMPLE

The **explain_lchmod_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_lchmod_or_die(pathname, mode);
```

## SEE ALSO

*lchmod*(2)
: change permissions of a file

*explain_lchmod*(3)
: explain *lchmod*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_lchown − explain lchown(2) errors

## SYNOPSIS

#include <libexplain/lchown.h>

const char *explain_lchown(const char *pathname, int owner, int group);
const char *explain_errno_lchown(int errnum, const char *pathname, int owner, int group);
void explain_message_lchown(char *message, int message_size, const char *pathname, int owner, int group);
void explain_message_errno_lchown(char *message, int message_size, int errnum, const char *pathname, int owner, int group);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *lchown*(2) system call.

### explain_lchown

const char *explain_lchown(const char *pathname, int owner, int group);

The **explain_lchown** function is used to obtain an explanation of an error returned by the *lchown*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (lchown(pathname, owner, group) < 0)
{
    fprintf(stderr, "%s\n", explain_lchown(pathname, owner, group));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *lchown*(2) system call.

*owner*
: The original owner, exactly as passed to the *lchown*(2) system call.

*group*
: The original group, exactly as passed to the *lchown*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_lchown

const char *explain_errno_lchown(int errnum, const char *pathname, int owner, int group);

The **explain_errno_lchown** function is used to obtain an explanation of an error returned by the *lchown*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (lchown(pathname, owner, group) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_lchown(err,
        pathname, owner, group));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *lchown*(2) system call.

*owner* The original owner, exactly as passed to the *lchown*(2) system call.

*group* The original group, exactly as passed to the *lchown*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_lchown**

void explain_message_lchown(char *message, int message_size, const char *pathname, int owner, int group);

The **explain_message_lchown** function may be used to obtain an explanation of an error returned by the *lchown*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (lchown(pathname, owner, group) < 0)
{
    char message[3000];
    explain_message_lchown(message, sizeof(message),
        pathname, owner, group);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *lchown*(2) system call.

*owner* The original owner, exactly as passed to the *lchown*(2) system call.

*group* The original group, exactly as passed to the *lchown*(2) system call.

**explain_message_errno_lchown**

void explain_message_errno_lchown(char *message, int message_size, int errnum, const char *pathname, int owner, int group);

The **explain_message_errno_lchown** function may be used to obtain an explanation of an error returned by the *lchown*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (lchown(pathname, owner, group) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_lchown(message, sizeof(message), err,
```

```
            pathname, owner, group);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *lchown*(2) system call.

*owner* The original owner, exactly as passed to the *lchown*(2) system call.

*group* The original group, exactly as passed to the *lchown*(2) system call.

## SEE ALSO

*lchown*(2)
change ownership of a file

*explain_lchown_or_die*(3)
change ownership of a file and report errors

## COPYRIGHT

## NAME

explain_lchown_or_die − change ownership of a file and report errors

## SYNOPSIS

#include <libexplain/lchown.h>

void explain_lchown_or_die(const char *pathname, int owner, int group);

## DESCRIPTION

The **explain_lchown_or_die** function is used to call the *lchown*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_lchown*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_lchown_or_die(pathname, owner, group);
```

*pathname*
: The pathname, exactly as to be passed to the *lchown*(2) system call.

*owner*
: The owner, exactly as to be passed to the *lchown*(2) system call.

*group*
: The group, exactly as to be passed to the *lchown*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*lchown*(2)
: change ownership of a file

*explain_lchown*(3)
: explain *lchown*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

LGPG − GNU Lesser General Public License

## DESCRIPTION

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007



This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

### 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

### 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

### 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

### 3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.

**4. Combined Works.**

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d)

Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

**5. Combined Libraries.**

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

**6. Revised Versions of the GNU Lesser General Public License.**

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU

Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

## NAME

explain_link − explain link(2) errors

## SYNOPSIS

#include <libexplain/link.h>

const char *explain_link(const char *oldpath, const char *newpath);
const char *explain_errno_link(int errnum, const char *oldpath, const char *newpath);
void explain_message_link(char *message, int message_size, const char *oldpath, const char *newpath);
void explain_message_errno_link(char *message, int message_size, int errnum, const char *oldpath, const char *newpath);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *link*(2) system call.

### explain_link

const char *explain_link(const char *oldpath, const char *newpath);

The **explain_link** function is used to obtain an explanation of an error returned by the *link*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (link(oldpath, newpath) < 0)
{
    fprintf(stderr, "%s\n", explain_link(oldpath, newpath));
    exit(EXIT_FAILURE);
}
```

*oldpath* The original oldpath, exactly as passed to the *link*(2) system call.

*newpath* The original newpath, exactly as passed to the *link*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_link

const char *explain_errno_link(int errnum, const char *oldpath, const char *newpath);

The **explain_errno_link** function is used to obtain an explanation of an error returned by the *link*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (link(oldpath, newpath) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_link(err, oldpath, newpath));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*oldpath* The original oldpath, exactly as passed to the *link*(2) system call.

*newpath* The original newpath, exactly as passed to the *link*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_link**

void explain_message_link(char *message, int message_size, const char *oldpath, const char *newpath);

The **explain_message_link** function may be used to obtain an explanation of an error returned by the *link*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (link(oldpath, newpath) < 0)
{
    char message[3000];
    explain_message_link(message, sizeof(message), oldpath, newpath);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*oldpath* The original oldpath, exactly as passed to the *link*(2) system call.

*newpath* The original newpath, exactly as passed to the *link*(2) system call.

**explain_message_errno_link**

void explain_message_errno_link(char *message, int message_size, int errnum, const char *oldpath, const char *newpath);

The **explain_message_errno_link** function may be used to obtain an explanation of an error returned by the *link*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (link(oldpath, newpath) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_link(message, sizeof(message), err,
        oldpath, newpath);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*oldpath* The original oldpath, exactly as passed to the *link*(2) system call.

*newpath* The original newpath, exactly as passed to the *link*(2) system call.

## SEE ALSO

*link*(2) make a new name for a file

*explain_link_or_die*(3)
make a new name for a file and report errors

## COPYRIGHT

## NAME

explain_link_or_die − make a new name for a file and report errors

## SYNOPSIS

#include <libexplain/link.h>

void explain_link_or_die(const char *oldpath, const char *newpath);

## DESCRIPTION

The **explain_link_or_die** function is used to call the *link*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_link*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_link_or_die(oldpath, newpath);
```

*oldpath* The oldpath, exactly as to be passed to the *link*(2) system call.

*newpath* The newpath, exactly as to be passed to the *link*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*link*(2) make a new name for a file

*explain_link*(3)
explain *link*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_listen − explain listen(2) errors

## SYNOPSIS

#include <libexplain/listen.h>

const char *explain_listen(int fildes, int backlog);
const char *explain_errno_listen(int errnum, int fildes, int backlog);
void explain_message_listen(char *message, int message_size, int fildes, int backlog);
void explain_message_errno_listen(char *message, int message_size, int errnum, int fildes, int backlog);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *listen*(2) system call.

### explain_listen

const char *explain_listen(int fildes, int backlog);

The **explain_listen** function is used to obtain an explanation of an error returned by the *listen*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (listen(fildes, backlog) < 0)
{
    fprintf(stderr, "%s\n", explain_listen(fildes, backlog));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_listen_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *listen*(2) system call.

*backlog* The original backlog, exactly as passed to the *listen*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_listen

const char *explain_errno_listen(int errnum, int fildes, int backlog);

The **explain_errno_listen** function is used to obtain an explanation of an error returned by the *listen*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (listen(fildes, backlog) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_listen(err, fildes, backlog));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_listen_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *listen*(2) system call.

*backlog* The original backlog, exactly as passed to the *listen*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_listen**

void explain_message_listen(char *message, int message_size, int fildes, int backlog);

The **explain_message_listen** function may be used to obtain an explanation of an error returned by the *listen*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (listen(fildes, backlog) < 0)
{
    char message[3000];
    explain_message_listen(message, sizeof(message), fildes, backlog);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_listen_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *listen*(2) system call.

*backlog* The original backlog, exactly as passed to the *listen*(2) system call.

**explain_message_errno_listen**

void explain_message_errno_listen(char *message, int message_size, int errnum, int fildes, int backlog);

The **explain_message_errno_listen** function may be used to obtain an explanation of an error returned by the *listen*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (listen(fildes, backlog) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_listen(message, sizeof(message), err,
        fildes, backlog);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_listen_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *listen*(2) system call.

*backlog* The original backlog, exactly as passed to the *listen*(2) system call.

## SEE ALSO

*listen*(2) listen for connections on a socket

*explain_listen_or_die*(3)
listen for connections on a socket and report errors

## COPYRIGHT

## NAME

explain_listen_or_die − listen for connections on a socket and report errors

## SYNOPSIS

#include <libexplain/listen.h>

void explain_listen_or_die(int fildes, int backlog);

## DESCRIPTION

The **explain_listen_or_die** function is used to call the *listen*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_listen*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_listen_or_die(fildes, backlog);
```

*fildes* The fildes, exactly as to be passed to the *listen*(2) system call.

*backlog* The backlog, exactly as to be passed to the *listen*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*listen*(2) listen for connections on a socket

*explain_listen*(3)
explain *listen*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_lseek − explain lseek(2) errors

## SYNOPSIS

#include <libexplain/lseek.h>
const char *explain_lseek(int fildes, long long offset, int whence);
const char *explain_errno_lseek(int errnum, int fildes, long long offset, int whence);
void explain_message_lseek(char *message, int message_size, int fildes, long long offset, int whence);
void explain_message_errno_lseek(char *message, int message_size, int errnum, int fildes, long long offset, int whence);

## DESCRIPTION

These functions may be used to obtain explanations for *lseek*(2) errors.

### explain_lseek

const char *explain_lseek(int fildes, long long offset, int whence);

The explain_lseek function may be used to obtain a human readable explanation of what went wrong in an *lseek*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (lseek(fd, offset, whence) == (off_t)-1)
{
    fprintf(stderr, '%s0, explain_lseek(fd, offset, whence);
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *lseek*(2) system call.

*offset* The original offset, exactly as passed to the *lseek*(2) system call.

*whence* The original whence, exactly as passed to the *lseek*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_lseek

const char *explain_errno_lseek(int errnum, int fildes, long long offset, int whence);

The explain_errno_lseek function may be used to obtain a human readable explanation of what went wrong in an *lseek*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (lseek(fd, offset, whence) == (off_t)-1)
{
    int errnum = errno;
    fprintf(stderr, '%s0, explain_errno_lseek(fd, eernum, offset,
        whence);
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *lseek*(2) system call.
*offset* The original offset, exactly as passed to the *lseek*(2) system call.
*whence* The original whence, exactly as passed to the *lseek*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_lseek**

void explain_message_lseek(char *message, int message_size, int fildes, long long offset, int whence);

The explain_message_lseek function may be used to obtain a human readable explanation of what went wrong in an *lseek*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (lseek(fd, offset, whence) == (off_t)-1)
{
    char message[3000];
    explain_message_lseek(message, sizeof(message), fd, offset, whence);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *lseek*(2) system call.

*offset* The original offset, exactly as passed to the *lseek*(2) system call.

*whence* The original whence, exactly as passed to the *lseek*(2) system call.

**explain_message_errno_lseek**

void explain_message_errno_lseek(char *message, int message_size, int errnum, int fildes, long long offset, int whence);

The explain_message_errno_lseek function may be used to obtain a human readable explanation of what went wrong in an *lseek*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (lseek(fd, offset, whence) == (off_t)-1)
{
    char message[3000];
    int errnum = errno;
    explain_message_errno_lseek(message, sizeof(message), errnum, fd,
        offset, whence);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes*
: The original fildes, exactly as passed to the *lseek*(2) system call.

*offset*
: The orginal offset, exactly as passed to the *lseek*(2) system call.

*whence*
: The original whence, exactly as passed to the *lseek*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_lseek_or_die − reposition file offset and report errors

## SYNOPSIS

#include <libexplain/lseek.h>

long long explain_lseek_or_die(int fildes, long long offset, int whence);

## DESCRIPTION

The **explain_lseek_or_die** function is used to call the *lseek*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_lseek*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
long long result = explain_lseek_or_die(fildes, offset, whence);
```

*fildes* The fildes, exactly as to be passed to the *lseek*(2) system call.

*offset* The offset, exactly as to be passed to the *lseek*(2) system call.

*whence* The whence, exactly as to be passed to the *lseek*(2) system call.

Returns: On successful, returns the resulting offset location as measured in bytes from the beginning of the file. On failure, prints an explanation and exits.

## SEE ALSO

*lseek*(2) reposition file offset

*explain_lseek*(3)
explain *lseek*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_lstat − explain lstat(3) errors

## SYNOPSIS

#include <libexplain/lstat.h>
const char *explain_lstat(const char *pathname, const struct stat *buf);
void explain_message_lstat(char *message, int message_size, const char *pathname, const struct stat *buf);
const char *explain_errno_lstat(int errnum, const char *pathname, const struct stat *buf);
void explain_message_errno_lstat(char *message, int message_size, int errnum, const char *pathname, const struct stat *buf);

## DESCRIPTION

These functions may be used to obtains explanations for *lstat*(2) errors.

### explain_lstat

const char *explain_lstat(const char *pathname, const struct stat *buf);

The explain_lstat function is used to obtain an explanation of an error returned by the *lstat*(2) function. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (lstat(pathname, &buf) < 0)
{
    fprintf(stderr, '%s0, explain_lstat(pathname, &buf));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *lstat*(2) system call.

*buf*
: The original buf, exactly as passed to the *lstat*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_lstat

"const char *explain_errno_lstat(int errnum, const char *pathname, const struct stat *buf);

The explain_errno_lstat function is used to obtain an explanation of an error returned by the *lstat*(2) function. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (lstat(pathname, &buf) < 0)
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_lstat(err, pathname, &buf));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *lstat*(2) system call.

*buf* The original buf, exactly as passed to the *lstat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_lstat**

void explain_message_lstat(char *message, int message_size, const char *pathname, const struct stat *buf);

The explain_message_lstat function is used to obtain an explanation of an error returned by the *lstat*(2) function. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (lstat(pathname, &buf) < 0)
{
    char message[3000];
    explain_message_lstat(message, sizeof(message), pathname, &buf);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *lstat*(2) system call.

*buf* The original buf, exactly as passed to the *lstat*(2) system call.

**explain_message_errno_lstat**

void explain_message_errno_lstat(char *message, int message_size, int errnum, const char *pathname, const struct stat *buf);

The explain_message_errno_lstat function is used to obtain an explanation of an error returned by the *lstat*(2) function. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (lstat(pathname, &buf) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_lstat(message, sizeof(message), err,
        pathname, &buf);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *lstat*(2) system call.

*buf*
: The original buf, exactly as passed to the *lstat*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_lstat_or_die − get file status and report errors

## SYNOPSIS

#include <libexplain/lstat.h>

void explain_lstat_or_die(const char *pathname, struct stat *buf);

## DESCRIPTION

The **explain_lstat_or_die** function is used to call the *lstat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_lstat*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_lstat_or_die(pathname , buf);
```

*pathname*
: The pathname, exactly as to be passed to the *lstat*(2) system call.

*buf*
: The buf, exactly as to be passed to the *lstat*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*lstat*(2) get file status

*explain_lstat*(3)
: explain *lstat*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_malloc − explain malloc(3) errors

## SYNOPSIS

#include <libexplain/malloc.h>

const char *explain_malloc(size_t size);
const char *explain_errno_malloc(int errnum, size_t size);
void explain_message_malloc(char *message, int message_size, size_t size);
void explain_message_errno_malloc(char *message, int message_size, int errnum, size_t size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *malloc*(3) system call.

### explain_malloc

const char *explain_malloc(size_t size);

The **explain_malloc** function is used to obtain an explanation of an error returned by the *malloc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (malloc(size) < 0)
{
    fprintf(stderr, "%s\n", explain_malloc(size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_malloc_or_die*(3) function.

*size* The original size, exactly as passed to the *malloc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_malloc

const char *explain_errno_malloc(int errnum, size_t size);

The **explain_errno_malloc** function is used to obtain an explanation of an error returned by the *malloc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (malloc(size) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_malloc(err, size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_malloc_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*size* The original size, exactly as passed to the *malloc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_malloc**

void explain_message_malloc(char *message, int message_size, size_t size);

The **explain_message_malloc** function may be used to obtain an explanation of an error returned by the *malloc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (malloc(size) < 0)
{
    char message[3000];
    explain_message_malloc(message, sizeof(message), size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_malloc_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*size* The original size, exactly as passed to the *malloc*(3) system call.

**explain_message_errno_malloc**

void explain_message_errno_malloc(char *message, int message_size, int errnum, size_t size);

The **explain_message_errno_malloc** function may be used to obtain an explanation of an error returned by the *malloc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (malloc(size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_malloc(message, sizeof(message), err, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_malloc_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*size* The original size, exactly as passed to the *malloc*(3) system call.

## SEE ALSO

*malloc*(3)
Allocate and free dynamic memory

*explain_malloc_or_die*(3)
Allocate and free dynamic memory and report errors

## COPYRIGHT

## NAME

explain_malloc_or_die − Allocate and free dynamic memory and report errors

## SYNOPSIS

#include <libexplain/malloc.h>

void *explain_malloc_or_die(size_t size);

## DESCRIPTION

The **explain_malloc_or_die** function is used to call the *malloc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_malloc*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
void *result = explain_malloc_or_die(size);
```

*size* The size, exactly as to be passed to the *malloc*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*malloc*(3)
Allocate and free dynamic memory

*explain_malloc*(3)
explain *malloc*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_mkdir − explain mkdir(2) errors

## SYNOPSIS

#include <libexplain/mkdir.h>

const char *explain_mkdir(const char *pathname);
const char *explain_errno_mkdir(int errnum, const char *pathname);
void explain_message_mkdir(char *message, int message_size, const char *pathname);
void explain_message_errno_mkdir(char *message, int message_size, int errnum, const char *pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *mkdir*(2) system call.

### explain_mkdir

const char *explain_mkdir(const char *pathname);

The **explain_mkdir** function is used to obtain an explanation of an error returned by the *mkdir*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (mkdir(pathname) < 0)
{
    fprintf(stderr, "%s\n", explain_mkdir(pathname));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *mkdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_mkdir

const char *explain_errno_mkdir(int errnum, const char *pathname);

The **explain_errno_mkdir** function is used to obtain an explanation of an error returned by the *mkdir*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (mkdir(pathname) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_mkdir(err, pathname));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *mkdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_mkdir**

void explain_message_mkdir(char *message, int message_size, const char *pathname);

The **explain_message_mkdir** function may be used to obtain an explanation of an error returned by the *mkdir*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (mkdir(pathname) < 0)
{
    char message[3000];
    explain_message_mkdir(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *mkdir*(2) system call.

**explain_message_errno_mkdir**

void explain_message_errno_mkdir(char *message, int message_size, int errnum, const char *pathname);

The **explain_message_errno_mkdir** function may be used to obtain an explanation of an error returned by the *mkdir*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (mkdir(pathname) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_mkdir(message, sizeof(message), err, pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *mkdir*(2) system call.

## SEE ALSO

*mkdir*(2) create a directory

*explain_mkdir_or_die*(3)
create a directory and report errors

## COPYRIGHT

## NAME

explain_mkdir_or_die − create a directory and report errors

## SYNOPSIS

#include <libexplain/mkdir.h>

void explain_mkdir_or_die(const char *pathname);

## DESCRIPTION

The **explain_mkdir_or_die** function is used to call the *mkdir*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_mkdir*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_mkdir_or_die(pathname);
```

*pathname*
: The pathname, exactly as to be passed to the *mkdir*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*mkdir*(2) create a directory

*explain_mkdir*(3)
: explain *mkdir*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_mkdtemp − explain *mkdtemp*(3) errors

## SYNOPSIS

#include <libexplain/mkdtemp.h>

const char *explain_mkdtemp(char *pathname);
const char *explain_errno_mkdtemp(int errnum, char *pathname);
void explain_message_mkdtemp(char *message, int message_size, char *pathname);
void explain_message_errno_mkdtemp(char *message, int message_size, int errnum, char *pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *mkdtemp*(3) system call.

### explain_mkdtemp

const char *explain_mkdtemp(char *pathname);

The **explain_mkdtemp** function is used to obtain an explanation of an error returned by the *mkdtemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *mkdtemp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = mkdtemp(pathname);
if (!result)
{
   fprintf(stderr, "%s\n", explain_mkdtemp(pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkdtemp_or_die*(3) function.

### explain_errno_mkdtemp

const char *explain_errno_mkdtemp(int errnum, char *pathname);

The **explain_errno_mkdtemp** function is used to obtain an explanation of an error returned by the *mkdtemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *mkdtemp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = mkdtemp(pathname);
if (!result)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_mkdtemp(err, pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkdtemp_or_die*(3) function.

**explain_message_mkdtemp**

void explain_message_mkdtemp(char *message, int message_size, char *pathname);

The **explain_message_mkdtemp** function is used to obtain an explanation of an error returned by the *mkdtemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *mkdtemp*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = mkdtemp(pathname);
if (!result)
{
    char message[3000];
    explain_message_mkdtemp(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkdtemp_or_die*(3) function.

**explain_message_errno_mkdtemp**

void explain_message_errno_mkdtemp(char *message, int message_size, int errnum, char *pathname);

The **explain_message_errno_mkdtemp** function is used to obtain an explanation of an error returned by the *mkdtemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *mkdtemp*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = mkdtemp(pathname);
if (!result)
```

```
{
    int err = errno;
    char message[3000];
   explain_message_errno_mkdtemp(message, sizeof(message), err,
   pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkdtemp_or_die*(3) function.

## SEE ALSO

*mkdtemp*(3)
: create a unique temporary directory

*explain_mkdtemp_or_die*(3)
: create a unique temporary directory and report errors

## COPYRIGHT

## NAME

explain_mkdtemp_or_die − create a unique temporary directory and report errors

## SYNOPSIS

#include <libexplain/mkdtemp.h>

char *explain_mkdtemp_or_die(char *pathname);
char *explain_mkdtemp_on_error(char *pathname);

## DESCRIPTION

The **explain_mkdtemp_or_die** function is used to call the *mkdtemp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mkdtemp*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_mkdtemp_on_error** function is used to call the *mkdtemp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mkdtemp*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *mkdtemp*(3) system call.

## RETURN VALUE

The **explain_mkdtemp_or_die** function only returns on success, see *mkdtemp*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_mkdtemp_on_error** function always returns the value return by the wrapped *mkdtemp*(3) system call.

## EXAMPLE

The **explain_mkdtemp_or_die** function is intended to be used in a fashion similar to the following example:

```
char *result = explain_mkdtemp_or_die(pathname);
```

## SEE ALSO

*mkdtemp*(3)
: create a unique temporary directory

*explain_mkdtemp*(3)
: explain *mkdtemp*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_mknod − explain *mknod*(2) errors

## SYNOPSIS

#include <libexplain/mknod.h>

const char *explain_mknod(const char *pathname, mode_t mode, dev_t dev);
const char *explain_errno_mknod(int errnum, const char *pathname, mode_t mode, dev_t dev);
void explain_message_mknod(char *message, int message_size, const char *pathname, mode_t mode, dev_t dev);
void explain_message_errno_mknod(char *message, int message_size, int errnum, const char *pathname, mode_t mode, dev_t dev);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *mknod*(2) system call.

### explain_mknod

const char *explain_mknod(const char *pathname, mode_t mode, dev_t dev);

The **explain_mknod** function is used to obtain an explanation of an error returned by the *mknod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *mknod*(2) system call.

*mode*
: The original mode, exactly as passed to the *mknod*(2) system call.

*dev*
: The original dev, exactly as passed to the *mknod*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (mknod(pathname, mode, dev) < 0)
{
    fprintf(stderr, "%s\n", explain_mknod(pathname, mode, dev));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mknod_or_die*(3) function.

### explain_errno_mknod

const char *explain_errno_mknod(int errnum, const char *pathname, mode_t mode, dev_t dev);

The **explain_errno_mknod** function is used to obtain an explanation of an error returned by the *mknod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *mknod*(2) system call.

*mode*
: The original mode, exactly as passed to the *mknod*(2) system call.

*dev* The original dev, exactly as passed to the *mknod*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (mknod(pathname, mode, dev) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_mknod(err, pathname,
   mode, dev));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mknod_or_die*(3) function.

**explain_message_mknod**

void explain_message_mknod(char *message, int message_size, const char *pathname, mode_t mode, dev_t dev);

The **explain_message_mknod** function is used to obtain an explanation of an error returned by the *mknod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *mknod*(2) system call.

*mode* The original mode, exactly as passed to the *mknod*(2) system call.

*dev* The original dev, exactly as passed to the *mknod*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (mknod(pathname, mode, dev) < 0)
{
    char message[3000];
   explain_message_mknod(message, sizeof(message), pathname,
   mode, dev);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mknod_or_die*(3) function.

**explain_message_errno_mknod**

void explain_message_errno_mknod(char *message, int message_size, int errnum, const char *pathname, mode_t mode, dev_t dev);

The **explain_message_errno_mknod** function is used to obtain an explanation of an error returned by the *mknod*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *mknod*(2) system call.

*mode* The original mode, exactly as passed to the *mknod*(2) system call.

*dev* The original dev, exactly as passed to the *mknod*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (mknod(pathname, mode, dev) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_mknod(message, sizeof(message), err,
   pathname, mode, dev);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mknod_or_die*(3) function.

## SEE ALSO

*mknod*(2)
create a special or ordinary file

*explain_mknod_or_die*(3)
create a special or ordinary file and report errors

## COPYRIGHT

## NAME

explain_mknod_or_die − create a special or ordinary file and report errors

## SYNOPSIS

#include <libexplain/mknod.h>

void explain_mknod_or_die(const char *pathname, mode_t mode, dev_t dev);
int explain_mknod_on_error(const char *pathname, mode_t mode, dev_t dev);

## DESCRIPTION

The **explain_mknod_or_die** function is used to call the *mknod*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mknod*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_mknod_on_error** function is used to call the *mknod*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mknod*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *mknod*(2) system call.

*mode*
: The mode, exactly as to be passed to the *mknod*(2) system call.

*dev*
: The dev, exactly as to be passed to the *mknod*(2) system call.

## RETURN VALUE

The **explain_mknod_or_die** function only returns on success, see *mknod*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_mknod_on_error** function always returns the value return by the wrapped *mknod*(2) system call.

## EXAMPLE

The **explain_mknod_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_mknod_or_die(pathname, mode, dev);
```

## SEE ALSO

*mknod*(2)
: create a special or ordinary file

*explain_mknod*(3)
: explain *mknod*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_mkostemp − explain *mkostemp*(3) errors

## SYNOPSIS

#include <libexplain/mkostemp.h>

const char *explain_mkostemp(char *templat, int flags);
const char *explain_errno_mkostemp(int errnum, char *templat, int flags);
void explain_message_mkostemp(char *message, int message_size, char *templat, int flags);
void explain_message_errno_mkostemp(char *message, int message_size, int errnum, char *templat, int flags);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *mkostemp*(3) system call.

### explain_mkostemp

const char *explain_mkostemp(char *templat, int flags);

The **explain_mkostemp** function is used to obtain an explanation of an error returned by the *mkostemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*templat* The original template, exactly as passed to the *mkostemp*(3) system call.

*flags* The original flags, exactly as passed to the *mkostemp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = mkostemp(templat, flags);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_mkostemp(templat, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkostemp_or_die*(3) function.

### explain_errno_mkostemp

const char *explain_errno_mkostemp(int errnum, char *templat, int flags);

The **explain_errno_mkostemp** function is used to obtain an explanation of an error returned by the *mkostemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*templat* The original template, exactly as passed to the *mkostemp*(3) system call.

*flags* The original flags, exactly as passed to the *mkostemp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = mkostemp(templat, flags);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_mkostemp(err, templat,
   flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkostemp_or_die*(3) function.

**explain_message_mkostemp**

void explain_message_mkostemp(char *message, int message_size, char *templat, int flags);

The **explain_message_mkostemp** function is used to obtain an explanation of an error returned by the *mkostemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*templat* The original template, exactly as passed to the *mkostemp*(3) system call.

*flags* The original flags, exactly as passed to the *mkostemp*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = mkostemp(templat, flags);
if (result < 0)
{
    char message[3000];
   explain_message_mkostemp(message, sizeof(message), templat,
   flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkostemp_or_die*(3) function.

**explain_message_errno_mkostemp**

void explain_message_errno_mkostemp(char *message, int message_size, int errnum, char *templat, int flags);

The **explain_message_errno_mkostemp** function is used to obtain an explanation of an error returned by the *mkostemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*templat* The original template, exactly as passed to the *mkostemp*(3) system call.

*flags* The original flags, exactly as passed to the *mkostemp*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = mkostemp(templat, flags);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_mkostemp(message, sizeof(message), err,
   templat, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkostemp_or_die*(3) function.

## SEE ALSO

*mkostemp*(3)
create a unique temporary file

*explain_mkostemp_or_die*(3)
create a unique temporary file and report errors

## COPYRIGHT

## NAME

explain_mkostemp_or_die − create a unique temporary file and report errors

## SYNOPSIS

#include <libexplain/mkostemp.h>

int explain_mkostemp_or_die(char *templat, int flags);
int explain_mkostemp_on_error(char *templat, int flags);

## DESCRIPTION

The **explain_mkostemp_or_die** function is used to call the *mkostemp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mkostemp*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_mkostemp_on_error** function is used to call the *mkostemp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mkostemp*(3) function, but still returns to the caller.

*templat* The template, exactly as to be passed to the *mkostemp*(3) system call.

*flags* The flags, exactly as to be passed to the *mkostemp*(3) system call.

## RETURN VALUE

The **explain_mkostemp_or_die** function only returns on success, see *mkostemp*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_mkostemp_on_error** function always returns the value return by the wrapped *mkostemp*(3) system call.

## EXAMPLE

The **explain_mkostemp_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_mkostemp_or_die(templat, flags);
```

## SEE ALSO

*mkostemp*(3)
create a unique temporary file

*explain_mkostemp*(3)
explain *mkostemp*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_mkstemp − explain *mkstemp*(3) errors

## SYNOPSIS

#include <libexplain/mkstemp.h>

const char *explain_mkstemp(char *templat);
const char *explain_errno_mkstemp(int errnum, char *templat);
void explain_message_mkstemp(char *message, int message_size, char *templat);
void explain_message_errno_mkstemp(char *message, int message_size, int errnum, char *templat);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *mkstemp*(3) system call.

### explain_mkstemp

const char *explain_mkstemp(char *templat);

The **explain_mkstemp** function is used to obtain an explanation of an error returned by the *mkstemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*templat* The original template, exactly as passed to the *mkstemp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = mkstemp(templat);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_mkstemp(templat));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkstemp_or_die*(3) function.

### explain_errno_mkstemp

const char *explain_errno_mkstemp(int errnum, char *templat);

The **explain_errno_mkstemp** function is used to obtain an explanation of an error returned by the *mkstemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*templat* The original template, exactly as passed to the *mkstemp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = mkstemp(templat);
```

```
if (result < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_mkstemp(err, templat));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkstemp_or_die*(3) function.

**explain_message_mkstemp**

void explain_message_mkstemp(char *message, int message_size, char *templat);

The **explain_message_mkstemp** function is used to obtain an explanation of an error returned by the *mkstemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*templat* The original template, exactly as passed to the *mkstemp*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = mkstemp(templat);
if (result < 0)
{
    char message[3000];
    explain_message_mkstemp(message, sizeof(message), templat);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mkstemp_or_die*(3) function.

**explain_message_errno_mkstemp**

void explain_message_errno_mkstemp(char *message, int message_size, int errnum, char *templat);

The **explain_message_errno_mkstemp** function is used to obtain an explanation of an error returned by the *mkstemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*templat* The original template, exactly as passed to the *mkstemp*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = mkstemp(templat);
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_mkstemp(message, sizeof(message), err,
```

```
                templat);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_mkstemp_or_die*(3) function.

## SEE ALSO

*mkstemp*(3)
create a unique temporary file

*explain_mkstemp_or_die*(3)
create a unique temporary file and report errors

## COPYRIGHT

## NAME

explain_mkstemp_or_die − create a unique temporary file and report errors

## SYNOPSIS

#include <libexplain/mkstemp.h>

int explain_mkstemp_or_die(char *templat);
int explain_mkstemp_on_error(char *templat);

## DESCRIPTION

The **explain_mkstemp_or_die** function is used to call the *mkstemp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mkstemp*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_mkstemp_on_error** function is used to call the *mkstemp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mkstemp*(3) function, but still returns to the caller.

*templat* The template, exactly as to be passed to the *mkstemp*(3) system call.

## RETURN VALUE

The **explain_mkstemp_or_die** function only returns on success, see *mkstemp*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_mkstemp_on_error** function always returns the value return by the wrapped *mkstemp*(3) system call.

## EXAMPLE

The **explain_mkstemp_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_mkstemp_or_die(templat);
```

## SEE ALSO

*mkstemp*(3)
create a unique temporary file

*explain_mkstemp*(3)
explain *mkstemp*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_mktemp − explain *mktemp*(3) errors

## SYNOPSIS

#include <libexplain/mktemp.h>

const char *explain_mktemp(char *pathname);
const char *explain_errno_mktemp(int errnum, char *pathname);
void explain_message_mktemp(char *message, int message_size, char *pathname);
void explain_message_errno_mktemp(char *message, int message_size, int errnum, char *pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *mktemp*(3) system call.

### explain_mktemp

const char *explain_mktemp(char *pathname);

The **explain_mktemp** function is used to obtain an explanation of an error returned by the *mktemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *mktemp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = mktemp(pathname);
if (!result)
{
   fprintf(stderr, "%s\n", explain_mktemp(pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mktemp_or_die*(3) function.

### explain_errno_mktemp

const char *explain_errno_mktemp(int errnum, char *pathname);

The **explain_errno_mktemp** function is used to obtain an explanation of an error returned by the *mktemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *mktemp*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = mktemp(pathname);
if (!result)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_mktemp(err, pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mktemp_or_die*(3) function.

**explain_message_mktemp**

void explain_message_mktemp(char *message, int message_size, char *pathname);

The **explain_message_mktemp** function is used to obtain an explanation of an error returned by the *mktemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *mktemp*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = mktemp(pathname);
if (!result)
{
    char message[3000];
    explain_message_mktemp(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mktemp_or_die*(3) function.

**explain_message_errno_mktemp**

void explain_message_errno_mktemp(char *message, int message_size, int errnum, char *pathname);

The **explain_message_errno_mktemp** function is used to obtain an explanation of an error returned by the *mktemp*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *mktemp*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = mktemp(pathname);
if (!result)
```

```
{
    int err = errno;
    char message[3000];
   explain_message_errno_mktemp(message, sizeof(message), err,
   pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mktemp_or_die*(3) function.

## SEE ALSO

*mktemp*(3)
: make a unique temporary filename

*explain_mktemp_or_die*(3)
: make a unique temporary filename and report errors

## COPYRIGHT

## NAME

explain_mktemp_or_die − make a unique temporary filename and report errors

## SYNOPSIS

#include <libexplain/mktemp.h>

char *explain_mktemp_or_die(char *pathname);
char *explain_mktemp_on_error(char *pathname);

## DESCRIPTION

The **explain_mktemp_or_die** function is used to call the *mktemp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mktemp*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_mktemp_on_error** function is used to call the *mktemp*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mktemp*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *mktemp*(3) system call.

## RETURN VALUE

The **explain_mktemp_or_die** function only returns on success, see *mktemp*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_mktemp_on_error** function always returns the value return by the wrapped *mktemp*(3) system call.

## EXAMPLE

The **explain_mktemp_or_die** function is intended to be used in a fashion similar to the following example:

```
char *result = explain_mktemp_or_die(pathname);
```

## SEE ALSO

*mktemp*(3)
: make a unique temporary filename

*explain_mktemp*(3)
: explain *mktemp*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_mmap − explain *mmap*(2) errors

## SYNOPSIS

#include <libexplain/mmap.h>

const char *explain_mmap(void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);
const char *explain_errno_mmap(int errnum, void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);
void explain_message_mmap(char *message, int message_size, void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);
void explain_message_errno_mmap(char *message, int message_size, int errnum, void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *mmap*(2) system call.

### explain_mmap

const char *explain_mmap(void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);

The **explain_mmap** function is used to obtain an explanation of an error returned by the *mmap*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *mmap*(2) system call.

*data_size*
The original data_size, exactly as passed to the *mmap*(2) system call.

*prot* The original prot, exactly as passed to the *mmap*(2) system call.

*flags* The original flags, exactly as passed to the *mmap*(2) system call.

*fildes* The original fildes, exactly as passed to the *mmap*(2) system call.

*offset* The original offset, exactly as passed to the *mmap*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void *result = mmap(data, data_size, prot, flags, fildes, offset);
if (!result)
{
   fprintf(stderr, "%s\n", explain_mmap(data, data_size, prot,
   flags, fildes, offset));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mmap_or_die*(3) function.

### explain_errno_mmap

const char *explain_errno_mmap(int errnum, void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);

The **explain_errno_mmap** function is used to obtain an explanation of an error returned by the *mmap*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *mmap*(2) system call.

*data_size*
The original data_size, exactly as passed to the *mmap*(2) system call.

*prot* The original prot, exactly as passed to the *mmap*(2) system call.

*flags* The original flags, exactly as passed to the *mmap*(2) system call.

*fildes* The original fildes, exactly as passed to the *mmap*(2) system call.

*offset* The original offset, exactly as passed to the *mmap*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void *result = mmap(data, data_size, prot, flags, fildes, offset);
if (!result)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_mmap(err, data,
   data_size, prot, flags, fildes, offset));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mmap_or_die*(3) function.

**explain_message_mmap**

void explain_message_mmap(char *message, int message_size, void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);

The **explain_message_mmap** function is used to obtain an explanation of an error returned by the *mmap*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *mmap*(2) system call.

*data_size*
The original data_size, exactly as passed to the *mmap*(2) system call.

*prot* The original prot, exactly as passed to the *mmap*(2) system call.

*flags* The original flags, exactly as passed to the *mmap*(2) system call.

*fildes* The original fildes, exactly as passed to the *mmap*(2) system call.

*offset* The original offset, exactly as passed to the *mmap*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void *result = mmap(data, data_size, prot, flags, fildes, offset);
if (!result)
```

```
{
    char message[3000];
   explain_message_mmap(message, sizeof(message), data,
   data_size, prot, flags, fildes, offset);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mmap_or_die*(3) function.

**explain_message_errno_mmap**

void explain_message_errno_mmap(char *message, int message_size, int errnum, void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);

The **explain_message_errno_mmap** function is used to obtain an explanation of an error returned by the *mmap*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *mmap*(2) system call.

*data_size*
The original data_size, exactly as passed to the *mmap*(2) system call.

*prot* The original prot, exactly as passed to the *mmap*(2) system call.

*flags* The original flags, exactly as passed to the *mmap*(2) system call.

*fildes* The original fildes, exactly as passed to the *mmap*(2) system call.

*offset* The original offset, exactly as passed to the *mmap*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void *result = mmap(data, data_size, prot, flags, fildes, offset);
if (!result)
{
    int err = errno;
    char message[3000];
   explain_message_errno_mmap(message, sizeof(message), err,
   data, data_size, prot, flags, fildes, offset);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_mmap_or_die*(3) function.

## SEE ALSO

*mmap*(2)
map file or device into memory

*explain_mmap_or_die*(3)
map file or device into memory and report errors

## COPYRIGHT

## NAME

explain_mmap_or_die − map file or device into memory and report errors

## SYNOPSIS

#include <libexplain/mmap.h>

void *explain_mmap_or_die(void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);
void *explain_mmap_on_error(void *data, size_t data_size, int prot, int flags, int fildes, off_t offset);

## DESCRIPTION

The **explain_mmap_or_die** function is used to call the *mmap*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mmap*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_mmap_on_error** function is used to call the *mmap*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_mmap*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *mmap*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *mmap*(2) system call.

*prot* The prot, exactly as to be passed to the *mmap*(2) system call.

*flags* The flags, exactly as to be passed to the *mmap*(2) system call.

*fildes* The fildes, exactly as to be passed to the *mmap*(2) system call.

*offset* The offset, exactly as to be passed to the *mmap*(2) system call.

## RETURN VALUE

The **explain_mmap_or_die** function only returns on success, see *mmap*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_mmap_on_error** function always returns the value return by the wrapped *mmap*(2) system call.

## EXAMPLE

The **explain_mmap_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_mmap_or_die(data, data_size, prot, flags, fildes, offset);
```

## SEE ALSO

*mmap*(2)
map file or device into memory

*explain_mmap*(3)
explain *mmap*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_munmap − explain *munmap*(2) errors

## SYNOPSIS

#include <libexplain/munmap.h>

const char *explain_munmap(void *data, size_t data_size);
const char *explain_errno_munmap(int errnum, void *data, size_t data_size);
void explain_message_munmap(char *message, int message_size, void *data, size_t data_size);
void explain_message_errno_munmap(char *message, int message_size, int errnum, void *data, size_t data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *munmap*(2) system call.

### explain_munmap

const char *explain_munmap(void *data, size_t data_size);

The **explain_munmap** function is used to obtain an explanation of an error returned by the *munmap*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *munmap*(2) system call.

*data_size*
The original data_size, exactly as passed to the *munmap*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (munmap(data, data_size) < 0)
{
   fprintf(stderr, "%s\n", explain_munmap(data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_munmap_or_die*(3) function.

### explain_errno_munmap

const char *explain_errno_munmap(int errnum, void *data, size_t data_size);

The **explain_errno_munmap** function is used to obtain an explanation of an error returned by the *munmap*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *munmap*(2) system call.

*data_size*
The original data_size, exactly as passed to the *munmap*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (munmap(data, data_size) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_munmap(err, data,
   data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_munmap_or_die*(3) function.

**explain_message_munmap**

void explain_message_munmap(char *message, int message_size, void *data, size_t data_size);

The **explain_message_munmap** function is used to obtain an explanation of an error returned by the *munmap*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *munmap*(2) system call.

*data_size*
The original data_size, exactly as passed to the *munmap*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (munmap(data, data_size) < 0)
{
    char message[3000];
   explain_message_munmap(message, sizeof(message), data,
   data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_munmap_or_die*(3) function.

**explain_message_errno_munmap**

void explain_message_errno_munmap(char *message, int message_size, int errnum, void *data, size_t data_size);

The **explain_message_errno_munmap** function is used to obtain an explanation of an error returned by the *munmap*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *munmap*(2) system call.

*data_size*
The original data_size, exactly as passed to the *munmap*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (munmap(data, data_size) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_munmap(message, sizeof(message), err,
   data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_munmap_or_die*(3) function.

## SEE ALSO

*munmap*(2)
unmap a file or device from memory

*explain_munmap_or_die*(3)
unmap a file or device from memory and report errors

## COPYRIGHT

## NAME

explain_munmap_or_die − unmap a file or device from memory and report errors

## SYNOPSIS

#include <libexplain/munmap.h>

void explain_munmap_or_die(void *data, size_t data_size);
int explain_munmap_on_error(void *data, size_t data_size);

## DESCRIPTION

The **explain_munmap_or_die** function is used to call the *munmap*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_munmap*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_munmap_on_error** function is used to call the *munmap*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_munmap*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *munmap*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *munmap*(2) system call.

## RETURN VALUE

The **explain_munmap_or_die** function only returns on success, see *munmap*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_munmap_on_error** function always returns the value return by the wrapped *munmap*(2) system call.

## EXAMPLE

The **explain_munmap_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_munmap_or_die(data, data_size);
```

## SEE ALSO

*munmap*(2)
unmap a file or device from memory

*explain_munmap*(3)
explain *munmap*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_nice − explain nice(2) errors

## SYNOPSIS

#include <libexplain/nice.h>

const char *explain_nice(int inc);
const char *explain_errno_nice(int errnum, int inc);
void explain_message_nice(char *message, int message_size, int inc);
void explain_message_errno_nice(char *message, int message_size, int errnum, int inc);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *nice*(2) system call.

### explain_nice

const char *explain_nice(int inc);

The **explain_nice** function is used to obtain an explanation of an error returned by the *nice*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*inc* The original inc, exactly as passed to the *nice*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = nice(inc);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_nice(inc));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_nice_or_die*(3) function.

### explain_errno_nice

const char *explain_errno_nice(int errnum, int inc);

The **explain_errno_nice** function is used to obtain an explanation of an error returned by the *nice*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*inc* The original inc, exactly as passed to the *nice*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = nice(inc);
```

```
if (result < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_nice(err, inc));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_nice_or_die*(3) function.

**explain_message_nice**

void explain_message_nice(char *message, int message_size, int inc);

The **explain_message_nice** function is used to obtain an explanation of an error returned by the *nice*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*inc* The original inc, exactly as passed to the *nice*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = nice(inc);
if (result < 0)
{
    char message[3000];
    explain_message_nice(message, sizeof(message), inc);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_nice_or_die*(3) function.

**explain_message_errno_nice**

void explain_message_errno_nice(char *message, int message_size, int errnum, int inc);

The **explain_message_errno_nice** function is used to obtain an explanation of an error returned by the *nice*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*inc* The original inc, exactly as passed to the *nice*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = nice(inc);
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_nice(message, sizeof(message), err,
```

```
                inc);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_nice_or_die*(3) function.

## SEE ALSO

*nice*(2) change process priority

*explain_nice_or_die*(3)
change process priority and report errors

## COPYRIGHT

## NAME

explain_nice_or_die − change process priority and report errors

## SYNOPSIS

#include <libexplain/nice.h>

int explain_nice_or_die(int inc);
int explain_nice_on_error(int inc);

## DESCRIPTION

The **explain_nice_or_die** function is used to call the *nice*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_nice*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_nice_on_error** function is used to call the *nice*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_nice*(3) function, but still returns to the caller.

*inc* The inc, exactly as to be passed to the *nice*(2) system call.

## RETURN VALUE

The **explain_nice_or_die** function only returns on success, see *nice*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_nice_on_error** function always returns the value return by the wrapped *nice*(2) system call.

## EXAMPLE

The **explain_nice_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_nice_or_die(inc);
```

## SEE ALSO

*nice*(2) change process priority

*explain_nice*(3)
explain *nice*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_open − explain open(2) errors

## SYNOPSIS

#include <libexplain/open.h>
const char *explain_open(const char *pathname, int flags, int mode);
const char *explain_errno_open(int errnum, const char *pathname, int flags, int mode);
void explain_message_open(char *message, int message_size, const char *pathname, int flags, int mode);
void explain_message_errno_open(char *message, int message_size, int errnum, const char *pathname, int flags, int mode);

## DESCRIPTION

These functions may be used to obtains explanations for *open*(2) errors.

**explain_open(const char *pathname, int flags, int mode);**

const char *explain_open(const char *pathname, int flags, int mode);

The explain_open function is used to obtain an explanation of an error returned by the *open*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int fd = open(pathname, flags, mode);
if (fd < 0)
{
    fprintf(stderr, '%s0, explain_open(pathname, flags, mode));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *open*(2) system call.

*flags*
: The original flags, exactly as passed to the *open*(2) system call.

*mode*
: The original mode, exactly as passed to the *open*(2) system call (or zero if the original call didn't need a mode argument).

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_errno_open**

const char *explain_errno_open(int errnum, const char *pathname, int flags, int mode);

The explain_errno_open function is used to obtain an explanation of an error returned by the *open*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int fd = open(pathname, flags, mode);
if (fd < 0)
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_open(err, pathname,
        flags, mode));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *open*(2) system call.

*flags* The original flags, exactly as passed to the *open*(2) system call.

*mode* The original mode, exactly as passed to the *open*(2) system call (or zero if the original call didn't need a mode argument).

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_open**

void explain_message_open(char *message, int message_size, const char *pathname, int flags, int mode);

The explain_message_open function is used to obtain an explanation of an error returned by the *open*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int fd = open(pathname, flags, mode);
if (fd < 0)
{
    char message[3000];
    explain_message_open(message, sizeof(message), pathname, flags,
        mode);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *open*(2) system call.

*flags* The original flags, exactly as passed to the *open*(2) system call.

*mode* The original mode, exactly as passed to the *open*(2) system call (or zero if the original call didn't need a mode argument).

**explain_message_errno_open**

void explain_message_errno_open(char *message, int message_size, int errnum, const char *pathname, int flags, int mode);

The explain_message_errno_open function is used to obtain an explanation of an error returned by the *open*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following exameple:

```
int fd = open(pathname, flags, mode);
if (fd < 0)
{
```

```
        int err = errno;
        char message[3000];
        explain_message_errno_open(message, sizeof(message), err, pathname,
            flags, mode);
        fprintf(stderr, '%s0, message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *open*(2) system call.

*flags* The original flags, exactly as passed to the *open*(2) system call.

*mode* The original mode, exactly as passed to the *open*(2) system call (or zero if the original call didn't need a mode argument).

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_opendir − explain opendir(3) errors

## SYNOPSIS

const char *explain_opendir(const char *pathname);
const char *explain_errno_opendir(int errnum, const char *pathname); int errnum, const char *pathname);
void explain_message_opendir(char *message, int message_size,
void explain_message_errno_opendir(char *message, int message_size, const char *pathname);

## DESCRIPTION

These functions may be used to explain *opendir*(3) errors.

### explain_opendir

const char *explain_opendir(const char *pathname);

The explain_opendir function is used to obtain an explanation of an error returned by the *opendir*(3) function. The least the message will contain is the value of strerror(errno), but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
DIR *dp = opendir(pathname);
if (!dp)
{
    fprintf(stderr, "%s\n", explain_opendir(pathname));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *opendir*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_opendir

const char *explain_errno_opendir(int errnum, const char *pathname); int errnum, const char *pathname);

The explain_errno_opendir function is used to obtain an explanation of an error returned by the *opendir*(3) function. The least the message will contain is the value of strerror(errnum), but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
DIR *dp = opendir(pathname);
if (!dp)
{
    int errnum = errno;
    const char *message = explain_errno_opendir(errnum, pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *opendir*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_opendir**

void explain_message_opendir(char *message, int message_size, const char *pathname);

The explain_message_opendir function is used to obtain an explanation of an error returned by the *opendir*(3) function. The least the message will contain is the value of strerror(errno), but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
DIR *dp = opendir(pathname);
if (!dp)
{
    char message[3000];
    explain_message_opendir(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe if the buffer is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *opendir*(3) system call.

**explain_message_errno_opendir**

void explain_message_errno_opendir(char *message, int message_size, const char *pathname);

The explain_message_errno_opendir function is used to obtain an explanation of an error returned by the *opendir*(3) function. The least the message will contain is the value of strerror(errnum), but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
DIR *dp = opendir(pathname);
if (!dp);
{
    int err = errno;
    char message[3000];
    explain_message_errno_opendir(message, sizeof(message), err,
        pathname);
    fprintf(stderr, '%s\n', message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe if the buffer is thread safe.
*message_size* The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *opendir*(3) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_opendir_or_die − open a directory and report errors

## SYNOPSIS

#include <libexplain/opendir.h>

DIR *explain_opendir_or_die(const char *pathname);

## DESCRIPTION

The **explain_opendir_or_die** function is used to call the *opendir*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_opendir*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
DIR *dir = explain_opendir_or_die(pathname);
```

*pathname*
: The pathname, exactly as to be passed to the *opendir*(3) system call.

Returns: On success, a pointer to the directory stream. On failure, prints an explanation and exits, does not return.

## SEE ALSO

*opendir*(3)
: open a directory

*explain_opendir*(3)
: explain *opendir*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_open_or_die − open file and report errors

## SYNOPSIS

```
#include <fcntl.h>
#include <libexplain/open.h>

int explain_open_or_die(const char *pathname, int flags, int mode);
```

## DESCRIPTION

Given a pathname for a file, open() returns a file descriptor, a small, non-negative integer for use in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process. See *open*(2) for more information.

## RETURN VALUE

On success, the new file descriptor is returned.

On error, a description of the error is obtained via *explain_open*(3), and printed on *stderr*. The process is the terminated via a call to the `exit(EXIT_FAILURE)` function.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_output − output error messages

## SYNOPSIS

```
#include <libexplain/output.h>
```

## DESCRIPTION

These functions may be used to write error messages.

**explain_output_message**

```
void explain_output_message(const char *text);
```

The explain_output_message function is used to print text. It is printed via the registered output class, see *explain_output_register*(3) for how.

*text* The text of the message to be printed. It has not been wrapped (yet).

**explain_output_error**

```
void explain_output_error(const char *fmt, ...);
```

The explain_output_error function is used to print a formatted error message. The printing is done via the *explain_output_message*(3) function.

*fmt* The format text of the message to be printed. See *printf*(3) for more information.

**explain_output_error_and_die**

```
void explain_output_error_and_die(const char *fmt, ...);
```

The explain_output_error_and_die function is used to print text, and then terminate immediately. The printing is done via the *explain_output_message*(3) function, process termination is via the *explain_output_exit_failure*(3) function.

*fmt* The format text of the message to be printed. See printf(3) for more information.

**explain_output_warning**

```
void explain_output_warning(const char *fmt, ...);
```

The explain_output_warning function is used to print a formatted error message, including the word "warning". The printing is done via the *explain_output_message*(3) function.

*fmt* The format text of the message to be printed. See *printf*(3) for more information.

**explain_output_exit**

```
void explain_output_exit(int status);
```

The explain_output_exit function is used to terminate execution. It is executed via the registered output class, *explain_output_register*(3) for how.

*status* The exist status requested.

**explain_output_exit_failure**

```
void explain_output_exit_failure(void);
```

The explain_output_exit_failure function is used to terminate execution, with exit status EXIT_FAILURE. It is executed via the registered output class, see *explain_output_register*(3) for how.

**explain_option_hanging_indent_set**

```
void explain_option_hanging_indent_set(int columns);
```

The explain_option_hanging_indent_set function is used to cause the output wrapping to use hanging indents. By default no hanging indent is used, but this can sometimes obfuscate the end of one error message and the beginning of another. A hanging indent results in continuation lines starting with white space, similar to RFC822 headers.

This can be set using the "`hanging-indent=`*n*" string in the EXPLAIN_OPTIONS environment variable. See *explain*(3) for more information.

Using this function will override any environment variable setting.

*columns* The number of columns of hanging indent to be used. A value of 0 means no hanging indent (all lines flush with left margin). A common value to use is 4: it doesn't consume too much of each line, and it is a clear indent.

## OUTPUT REDIRECTION

It is possible to change how and where libexplain sends its output, and even how it calls the *exit*(2) function. This functionality is used by the `explain_*_or_die` and `explain_*_on_error` functions.

By default, libexplain will wrap and print error messages on stderr, and call the *exit*(2) system call to terminate execution.

Clients of the libexplain library may choose to use some message handling facilities provided by libexplain, or they may choose to implement their own.

**syslog**

To cause all output to be sent to syslog, use

```
explain_output_register(explain_output_syslog_new());
```

This is useful for servers and daemons.

**stderr and syslog**

The "tee" output class can be used to duplicate output. To cause all output to be sent to both stderr and syslog, use

```
explain_output_register
(
    explain_output_tee_new
    (
        explain_output_stderr_new(),
        explain_output_syslog_new()
    )
);
```

If you need more than two, use several instances of "tee", cascaded.

**stderr and a file**

To cause all output to be sent to both stderr and a regular file, use

```
explain_output_register
(
    explain_output_tee_new
    (
        explain_output_stderr_new(),
        explain_output_file_new(filename, 0)
    )
);
```

See the `<libexplain/output.h>` file for extensive documentation.

**explain_output_new**

```
explain_output_t *explain_output_new(const explain_output_vtable_t
*vtable);
```

The explain_output_new function may be used to create a new dynamically allocated instance of explain_output_t.

*vtable* The struct containing the pointers to the methods of the derived class.

*returns* NULL on error (i.e. malloc failed), or a pointer to a new dynamically allocated instance of the class.

**explain_output_stderr_new**

```
explain_output_t *explain_output_stderr_new(void);
```

The explain_output_stderr_new function may be used to create a new dynamically allocated instance of an explain_output_t class that writes to stderr, and exits via *exit*(2);

This is the default output handler.

*returns* NULL on error (i.e. malloc failed), or a pointer to a new dynamically allocated instance of the stderr class.

**explain_output_syslog_new**

```
explain_output_t *explain_output_syslog_new(void);
```

The explain_output_syslog_new function may be used to create a new dynamically allocated instance of an explain_output_t class that writes to syslog, and exits via *exit*(2);

The following values are used:

```
option = 0
facility = LOG_USER
level = LOG_ERR
```

See *syslog*(3) for more information.

*returns* NULL on error (i.e. *malloc*(3) failed), or a pointer to a new dynamically allocated instance of the syslog class.

**explain_output_syslog_new1**

```
explain_output_t *explain_output_syslog_new1(int level);
```

The explain_output_syslog_new1 function may be used to create a new dynamically allocated instance of an explain_output_t class that writes to syslog, and exits via *exit*(2);

The following values are used:

```
option = 0
facility = LOG_USER
```

See *syslog*(3) for more information.

*level* The syslog level to be used, see syslog(3) for a definition.

*returns* NULL on error (i.e. *malloc*(3) failed), or a pointer to a new dynamically allocated instance of the syslog class.

**explain_output_syslog_new3**

```
explain_output_t *explain_output_syslog_new3(int option, int facility,
int level);
```

The explain_output_syslog_new3 function may be used to create a new dynamically allocated instance of an explain_output_t class that writes to syslog, and exits via *exit*(2);

If you want different facilities or levels, create multiple instances.

*option* The syslog option to be used, see syslog(3) for a definition.

*facility* The syslog facility to be used, see syslog(3) for a definition.

*level* The syslog level to be used, see syslog(3) for a definition.

*returns* NULL on error (i.e. *malloc*(3) failed), or a pointer to a new dynamically allocated instance of the syslog class.

**explain_output_file_new**

```
explain_output_t *explain_output_file_new(const char *filename, int
append);
```

The explain_output_file_new function may be used to create a new dynamically allocated instance of an

explain_output_t class that writes to a file, and exits via *exit*(2).

*filename* The file to be opened and written to.

*append* true (non-zero) if messages are to be appended to the file, false (zero) if the file is to be replaced with new contents.

*returns* NULL on error (i.e. *malloc*(3) or *open*(2) failed), or a pointer to a new dynamically allocated instance of the syslog class.

**explain_output_tee_new**

```
explain_output_t *explain_output_tee_new(explain_output_t *first,
explain_output_t *second);
```

The explain_output_tee_new function may be used to create a new dynamically allocated instance of an explain_output_t class that writes to **two** other output classes.

*first* The first output class to write to.

*second* The second output class to write to.

*returns* NULL on error (i.e. *malloc*(3) failed), or a pointer to a new dynamically allocated instance of the syslog class.

The output subsystem will "own" the *first* and *second* objects after this call. You may not make any reference to these pointers ever again. The output subsystem will destroy these objects and free the memory when it feels like it.

**explain_output_register**

```
void explain_output_register(explain_output_t *op);
```

The explain_output_register function is used to change libexplain's default output handling facilities with something else. The NULL pointer restores libexplain's default processing.

If no output class is registered, the default is to wrap and print to stderr, and to exit via the *exit*(2) system call.

*op* Pointer to the explain_output_t instance to be operated on.

The output subsystem will "own" the pointer after this call. You may not make any reference to this pointer ever again. The output subsystem will destroy the object and free the memory when it feels like it.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_pathconf − explain pathconf(3) errors

## SYNOPSIS

#include <libexplain/pathconf.h>

const char *explain_pathconf(const char *pathname, int name);
const char *explain_errno_pathconf(int errnum, const char *pathname, int name);
void explain_message_pathconf(char *message, int message_size, const char *pathname, int name);
void explain_message_errno_pathconf(char *message, int message_size, int errnum, const char *pathname, int name);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *pathconf*(3) system call.

### explain_pathconf

const char *explain_pathconf(const char *pathname, int name);

The **explain_pathconf** function is used to obtain an explanation of an error returned by the *pathconf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (pathconf(pathname, name) < 0)
{
    fprintf(stderr, "%s\n", explain_pathconf(pathname, name));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pathconf_or_die*(3) function.

*pathname*
: The original pathname, exactly as passed to the *pathconf*(3) system call.

*name*
: The original name, exactly as passed to the *pathconf*(3) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_pathconf

const char *explain_errno_pathconf(int errnum, const char *pathname, int name);

The **explain_errno_pathconf** function is used to obtain an explanation of an error returned by the *pathconf*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (pathconf(pathname, name) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_pathconf(err, pathname, name));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pathconf_or_die*(3) function.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *pathconf*(3) system call.

*name* The original name, exactly as passed to the *pathconf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_pathconf**

void explain_message_pathconf(char *message, int message_size, const char *pathname, int name);

The **explain_message_pathconf** function may be used to obtain an explanation of an error returned by the *pathconf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (pathconf(pathname, name) < 0)
{
    char message[3000];
    explain_message_pathconf(message, sizeof(message), pathname, name);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pathconf_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *pathconf*(3) system call.

*name* The original name, exactly as passed to the *pathconf*(3) system call.

**explain_message_errno_pathconf**

void explain_message_errno_pathconf(char *message, int message_size, int errnum, const char *pathname, int name);

The **explain_message_errno_pathconf** function may be used to obtain an explanation of an error returned by the *pathconf*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (pathconf(pathname, name) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_pathconf(message, sizeof(message), err,
        pathname, name);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pathconf_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *pathconf*(3) system call.

*name* The original name, exactly as passed to the *pathconf*(3) system call.

## SEE ALSO

*pathconf*(3)
get configuration values for files

*explain_pathconf_or_die*(3)
get configuration values for files and report errors

## COPYRIGHT

## NAME

explain_pathconf_or_die − get configuration values and report errors

## SYNOPSIS

#include <libexplain/pathconf.h>

long explain_pathconf_or_die(const char *pathname, int name);

## DESCRIPTION

The **explain_pathconf_or_die** function is used to call the *pathconf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_pathconf*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
long value = explain_pathconf_or_die(pathname, name);
```

Note that a −1 return value is still possible, meaning the system does not have a limit for the requested resource.

*pathname*
: The pathname, exactly as to be passed to the *pathconf*(3) system call.

*name*
: The name, exactly as to be passed to the *pathconf*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*pathconf*(3)
: get configuration values for files

*explain_pathconf*(3)
: explain *pathconf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_pclose − explain pclose(3) errors

## SYNOPSIS

#include <libexplain/pclose.h>

const char *explain_pclose(FILE *fp);
const char *explain_errno_pclose(int errnum, FILE *fp);
void explain_message_pclose(char *message, int message_size, FILE *fp);
void explain_message_errno_pclose(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *pclose*(3) system call.

### explain_pclose

const char *explain_pclose(FILE *fp);

The **explain_pclose** function is used to obtain an explanation of an error returned by the *pclose*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (pclose(fp) < 0)
{
    fprintf(stderr, "%s\n", explain_pclose(fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pclose_or_die*(3) function.

*fp* The original fp, exactly as passed to the *pclose*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_pclose

const char *explain_errno_pclose(int errnum, FILE *fp);

The **explain_errno_pclose** function is used to obtain an explanation of an error returned by the *pclose*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (pclose(fp) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_pclose(err, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pclose_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *pclose*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_pclose**

void explain_message_pclose(char *message, int message_size, FILE *fp);

The **explain_message_pclose** function may be used to obtain an explanation of an error returned by the *pclose*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (pclose(fp) < 0)
{
    char message[3000];
    explain_message_pclose(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pclose_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *pclose*(3) system call.

**explain_message_errno_pclose**

void explain_message_errno_pclose(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_pclose** function may be used to obtain an explanation of an error returned by the *pclose*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (pclose(fp) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_pclose(message, sizeof(message), err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pclose_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *pclose*(3) system call.

## SEE ALSO

*pclose*(3)
process I/O

*explain_pclose_or_die*(3)
process I/O and report errors

## COPYRIGHT

## NAME

explain_pclose_or_die − process I/O and report errors

## SYNOPSIS

#include <libexplain/pclose.h>

int explain_pclose_or_die(FILE *fp);
int explain_pclose_success(FILE *fp);
void explain_pclose_success_or_die(FILE *fp);

## DESCRIPTION

These functions may be used to wait for program termination, and then reprt errors returned by the *pclose*(3) system call.

### explain_pclose_or_die

int explain_pclose_or_die(FILE *fp);

The **explain_pclose_or_die** function is used to call the *pclose*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_pclose*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int status = explain_pclose_or_die(fp);
```

*fp* The fp, exactly as to be passed to the *pclose*(3) system call.

Returns: This function only returns on success, see *pclose*(3) for more information. On failure, prints an explanation and exits.

### explain_pclose_success_or_die

void explain_pclose_success_or_die(FILE *);

The **explain_pclose_success_or_die** function is used to call the *pclose*(3) system call. On failure, including any exit status other than EXIT_SUCCESS, an explanation will be printed to *stderr*, obtained from *explain_pclose*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_pclose_success_or_die(fp);
```

*fp* The fp, exactly as to be passed to the *pclose*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

### explain_pclose_success

int explain_pclose_success(FILE *fp);

The **explain_pclose_success** function is used to call the *pclose*(3) system call. On failure, including any exit status other than EXIT_SUCCESS, an explanation will be printed to *stderr*, obtained from *explain_pclose*(3). However, the priniting of an error message does **not** also cause *exit*(2) to be called.

This function is intended to be used in a fashion similar to the following example:

```
int status = explain_pclose_success(command);
```

*fp* The fp, exactly as to be passed to the *pclose*(3) system call.

Returns: the value returned by the *pclose*(3) system call. In all cases other than EXIT_SUCCESS, an error message will also have been printed to stderr.

## SEE ALSO

*pclose*(3)
process I/O

*explain_pclose*(3)
explain *pclose*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_pipe − explain pipe(2) errors

## SYNOPSIS

#include <libexplain/pipe.h>

const char *explain_pipe(int *pipefd);
const char *explain_errno_pipe(int errnum, int *pipefd);
void explain_message_pipe(char *message, int message_size, int *pipefd);
void explain_message_errno_pipe(char *message, int message_size, int errnum, int *pipefd);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *pipe*(2) system call.

### explain_pipe

const char *explain_pipe(int *pipefd);

The **explain_pipe** function is used to obtain an explanation of an error returned by the *pipe*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (pipe(pipefd) < 0)
{
    fprintf(stderr, "%s\n", explain_pipe(pipefd));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pipe_or_die*(3) function.

*pipefd* The original pipefd, exactly as passed to the *pipe*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_pipe

const char *explain_errno_pipe(int errnum, int *pipefd);

The **explain_errno_pipe** function is used to obtain an explanation of an error returned by the *pipe*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (pipe(pipefd) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_pipe(err, pipefd));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pipe_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pipefd* The original pipefd, exactly as passed to the *pipe*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_pipe**

void explain_message_pipe(char *message, int message_size, int *pipefd);

The **explain_message_pipe** function may be used to obtain an explanation of an error returned by the *pipe*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (pipe(pipefd) < 0)
{
    char message[3000];
    explain_message_pipe(message, sizeof(message), pipefd);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pipe_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pipefd* The original pipefd, exactly as passed to the *pipe*(2) system call.

**explain_message_errno_pipe**

void explain_message_errno_pipe(char *message, int message_size, int errnum, int *pipefd);

The **explain_message_errno_pipe** function may be used to obtain an explanation of an error returned by the *pipe*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (pipe(pipefd) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_pipe(message, sizeof(message), err, pipefd);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pipe_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pipefd* The original pipefd, exactly as passed to the *pipe*(2) system call.

## SEE ALSO

*pipe*(2) create pipe

*explain_pipe_or_die*(3)
create pipe and report errors

## COPYRIGHT

## NAME

explain_pipe_or_die − create pipe and report errors

## SYNOPSIS

#include <libexplain/pipe.h>

void explain_pipe_or_die(int *pipefd);

## DESCRIPTION

The **explain_pipe_or_die** function is used to call the *pipe*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_pipe*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_pipe_or_die(pipefd);
```

*pipefd* The pipefd, exactly as to be passed to the *pipe*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*pipe*(2) create pipe

*explain_pipe*(3)
explain *pipe*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_poll − explain *poll*(2) errors

## SYNOPSIS

#include <libexplain/poll.h>

const char *explain_poll(struct pollfd *fds, int nfds, int timeout);
const char *explain_errno_poll(int errnum, struct pollfd *fds, int nfds, int timeout);
void explain_message_poll(char *message, int message_size, struct pollfd *fds, int nfds, int timeout);
void explain_message_errno_poll(char *message, int message_size, int errnum, struct pollfd *fds, int nfds, int timeout);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *poll*(2) system call.

### explain_poll

const char *explain_poll(struct pollfd *fds, int nfds, int timeout);

The **explain_poll** function is used to obtain an explanation of an error returned by the *poll*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fds* The original fds, exactly as passed to the *poll*(2) system call.

*nfds* The original nfds, exactly as passed to the *poll*(2) system call.

*timeout* The original timeout, exactly as passed to the *poll*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = poll(fds, nfds, timeout);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_poll(fds, nfds, timeout));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_poll_or_die*(3) function.

### explain_errno_poll

const char *explain_errno_poll(int errnum, struct pollfd *fds, int nfds, int timeout);

The **explain_errno_poll** function is used to obtain an explanation of an error returned by the *poll*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fds* The original fds, exactly as passed to the *poll*(2) system call.

*nfds* The original nfds, exactly as passed to the *poll*(2) system call.

*timeout* The original timeout, exactly as passed to the *poll*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = poll(fds, nfds, timeout);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_poll(err, fds, nfds,
   timeout));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_poll_or_die*(3) function.

**explain_message_poll**

void explain_message_poll(char *message, int message_size, struct pollfd *fds, int nfds, int timeout);

The **explain_message_poll** function is used to obtain an explanation of an error returned by the *poll*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fds* The original fds, exactly as passed to the *poll*(2) system call.

*nfds* The original nfds, exactly as passed to the *poll*(2) system call.

*timeout* The original timeout, exactly as passed to the *poll*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = poll(fds, nfds, timeout);
if (result < 0)
{
    char message[3000];
   explain_message_poll(message, sizeof(message), fds, nfds,
   timeout);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_poll_or_die*(3) function.

**explain_message_errno_poll**

void explain_message_errno_poll(char *message, int message_size, int errnum, struct pollfd *fds, int nfds, int timeout);

The **explain_message_errno_poll** function is used to obtain an explanation of an error returned by the *poll*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fds* The original fds, exactly as passed to the *poll*(2) system call.

*nfds* The original nfds, exactly as passed to the *poll*(2) system call.

*timeout* The original timeout, exactly as passed to the *poll*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = poll(fds, nfds, timeout);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_poll(message, sizeof(message), err, fds,
   nfds, timeout);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_poll_or_die*(3) function.

## SEE ALSO

*poll*(2) wait for some event on a file descriptor

*explain_poll_or_die*(3)
wait for some event on a file descriptor and report errors

## COPYRIGHT

## NAME

explain_poll_or_die − wait for some event on file descriptor and report errors

## SYNOPSIS

#include <libexplain/poll.h>

int explain_poll_or_die(struct pollfd *fds, int nfds, int timeout);
int explain_poll_on_error(struct pollfd *fds, int nfds, int timeout);

## DESCRIPTION

The **explain_poll_or_die** function is used to call the *poll*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_poll*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_poll_on_error** function is used to call the *poll*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_poll*(3) function, but still returns to the caller.

*fds* The fds, exactly as to be passed to the *poll*(2) system call.

*nfds* The nfds, exactly as to be passed to the *poll*(2) system call.

*timeout* The timeout, exactly as to be passed to the *poll*(2) system call.

## RETURN VALUE

The **explain_poll_or_die** function only returns on success, see *poll*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_poll_on_error** function always returns the value return by the wrapped *poll*(2) system call.

## EXAMPLE

The **explain_poll_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_poll_or_die(fds, nfds, timeout);
```

## SEE ALSO

*poll*(2) wait for some event on a file descriptor

*explain_poll*(3)
explain *poll*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_popen − explain popen(3) errors

## SYNOPSIS

#include <libexplain/popen.h>

const char *explain_popen(const char *command, const char *flags);
const char *explain_errno_popen(int errnum, const char *command, const char *flags);
void explain_message_popen(char *message, int message_size, const char *command, const char *flags);
void explain_message_errno_popen(char *message, int message_size, int errnum, const char *command, const char *flags);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *popen*(3) system call.

### explain_popen

const char *explain_popen(const char *command, const char *flags);

The **explain_popen** function is used to obtain an explanation of an error returned by the *popen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = popen(command, flags);
if (!fp)
{
    fprintf(stderr, "%s\n", explain_popen(command, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_popen_or_die*(3) function.

*command*
: The original command, exactly as passed to the *popen*(3) system call.

*flags*
: The original flags, exactly as passed to the *popen*(3) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_popen

const char *explain_errno_popen(int errnum, const char *command, const char *flags);

The **explain_errno_popen** function is used to obtain an explanation of an error returned by the *popen*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = popen(command, flags);
if (!fp)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_popen(err, command, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_popen_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*command*
The original command, exactly as passed to the *popen*(3) system call.

*flags* The original flags, exactly as passed to the *popen*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_popen**

void explain_message_popen(char *message, int message_size, const char *command, const char *flags);

The **explain_message_popen** function may be used to obtain an explanation of an error returned by the *popen*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = popen(command, flags);
if (!fp)
{
    char message[3000];
    explain_message_popen(message, sizeof(message), command, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_popen_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*command*
The original command, exactly as passed to the *popen*(3) system call.

*flags* The original flags, exactly as passed to the *popen*(3) system call.

**explain_message_errno_popen**

void explain_message_errno_popen(char *message, int message_size, int errnum, const char *command, const char *flags);

The **explain_message_errno_popen** function may be used to obtain an explanation of an error returned by the *popen*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = popen(command, flags);
if (!fp)
{
    int err = errno;
    char message[3000];
    explain_message_errno_popen(message, sizeof(message),
        err, command, flags);
```

```
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_popen_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*command*
The original command, exactly as passed to the *popen*(3) system call.

*flags* The original flags, exactly as passed to the *popen*(3) system call.

## SEE ALSO

*popen*(3)
process I/O

*explain_popen_or_die*(3)
process I/O and report errors

## COPYRIGHT

## NAME

explain_popen_or_die − process I/O and report errors

## SYNOPSIS

#include <libexplain/popen.h>

FILE *explain_popen_or_die(const char *command, const char *flags);

## DESCRIPTION

The **explain_popen_or_die** function is used to call the *popen*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_popen*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
FILE *fp = explain_popen_or_die(command, flags);
```

*command*
: The command, exactly as to be passed to the *popen*(3) system call.

*flags*
: The flags, exactly as to be passed to the *popen*(3) system call.

Returns:
: This function only returns on success, see *popen*(3) for more information. On failure, prints an explanation and exits.

## SEE ALSO

*popen*(3)
: process I/O

*explain_popen*(3)
: explain *popen*(3) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_pread − explain pread(2) errors

## SYNOPSIS

#include <libexplain/pread.h>

const char *explain_pread(int fildes, void *data, size_t data_size, off_t offset);
const char *explain_errno_pread(int errnum, int fildes, void *data, size_t data_size, off_t offset);
void explain_message_pread(char *message, int message_size, int fildes, void *data, size_t data_size, off_t offset);
void explain_message_errno_pread(char *message, int message_size, int errnum, int fildes, void *data, size_t data_size, off_t offset);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *pread*(2) system call.

### explain_pread

const char *explain_pread(int fildes, void *data, size_t data_size, off_t offset);

The **explain_pread** function is used to obtain an explanation of an error returned by the *pread*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *pread*(2) system call.

*data* The original data, exactly as passed to the *pread*(2) system call.

*data_size*
The original data_size, exactly as passed to the *pread*(2) system call.

*offset* The original offset, exactly as passed to the *pread*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = pread(fildes, data, data_size, offset);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_pread(fildes, data, data_size,
   offset));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pread_or_die*(3) function.

### explain_errno_pread

const char *explain_errno_pread(int errnum, int fildes, void *data, size_t data_size, off_t offset);

The **explain_errno_pread** function is used to obtain an explanation of an error returned by the *pread*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *pread*(2) system call.

*data* The original data, exactly as passed to the *pread*(2) system call.

*data_size*
The original data_size, exactly as passed to the *pread*(2) system call.

*offset* The original offset, exactly as passed to the *pread*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = pread(fildes, data, data_size, offset);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_pread(err, fildes, data,
   data_size, offset));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pread_or_die*(3) function.

**explain_message_pread**

void explain_message_pread(char *message, int message_size, int fildes, void *data, size_t data_size, off_t offset);

The **explain_message_pread** function is used to obtain an explanation of an error returned by the *pread*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *pread*(2) system call.

*data* The original data, exactly as passed to the *pread*(2) system call.

*data_size*
The original data_size, exactly as passed to the *pread*(2) system call.

*offset* The original offset, exactly as passed to the *pread*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = pread(fildes, data, data_size, offset);
if (result < 0)
{
    char message[3000];
   explain_message_pread(message, sizeof(message), fildes, data,
   data_size, offset);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pread_or_die*(3) function.

**explain_message_errno_pread**

void explain_message_errno_pread(char *message, int message_size, int errnum, int fildes, void *data, size_t data_size, off_t offset);

The **explain_message_errno_pread** function is used to obtain an explanation of an error returned by the *pread*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *pread*(2) system call.

*data* The original data, exactly as passed to the *pread*(2) system call.

*data_size*
The original data_size, exactly as passed to the *pread*(2) system call.

*offset* The original offset, exactly as passed to the *pread*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = pread(fildes, data, data_size, offset);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_pread(message, sizeof(message), err,
   fildes, data, data_size, offset);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pread_or_die*(3) function.

## SEE ALSO

*pread*(2) read from or write to a file descriptor at a given offset

*explain_pread_or_die*(3)
read from or write to a file descriptor at a given offset and report errors

## COPYRIGHT

## NAME

explain_pread_or_die − seek and read from a file descriptor and report errors

## SYNOPSIS

#include <libexplain/pread.h>

ssize_t explain_pread_or_die(int fildes, void *data, size_t data_size, off_t offset);
ssize_t explain_pread_on_error(int fildes, void *data, size_t data_size, off_t offset))

## DESCRIPTION

The **explain_pread_or_die** function is used to call the *pread*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_pread*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_pread_on_error** function is used to call the *pread*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_pread*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *pread*(2) system call.

*data* The data, exactly as to be passed to the *pread*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *pread*(2) system call.

*offset* The offset, exactly as to be passed to the *pread*(2) system call.

## RETURN VALUE

The **explain_pread_or_die** function only returns on success, see *pread*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_pread_on_error** function always returns the value return by the wrapped *pread*(2) system call.

## EXAMPLE

The **explain_pread_or_die** function is intended to be used in a fashion similar to the following example:

```
ssize_t result = explain_pread_or_die(fildes, data, data_size, offset);
```

## SEE ALSO

*pread*(2) read from a file descriptor at a given offset

*explain_pread*(3)
explain *pread*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_printf − explain *printf*(3) errors

## SYNOPSIS

#include <libexplain/printf.h>

const char *explain_printf(const char *format);
const char *explain_errno_printf(int errnum, const char *format);
void explain_message_printf(char *message, int message_size, const char *format);
void explain_message_errno_printf(char *message, int message_size, int errnum, const char *format);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *printf*(3) system call.

### explain_printf

const char *explain_printf(const char *format);

The **explain_printf** function is used to obtain an explanation of an error returned by the *printf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*format* The original format, exactly as passed to the *printf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = printf(format);
if (result < 0 && errno != 0)
{
   fprintf(stderr, "%s\n", explain_printf(format));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_printf_or_die*(3) function.

### explain_errno_printf

const char *explain_errno_printf(int errnum, const char *format);

The **explain_errno_printf** function is used to obtain an explanation of an error returned by the *printf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*format* The original format, exactly as passed to the *printf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = printf(format);
if (result < 0 && errno != 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_printf(err, format));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_printf_or_die*(3) function.

**explain_message_printf**

void explain_message_printf(char *message, int message_size, const char *format);

The **explain_message_printf** function is used to obtain an explanation of an error returned by the *printf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*format* The original format, exactly as passed to the *printf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = printf(format);
if (result < 0 && errno != 0)
{
    char message[3000];
   explain_message_printf(message, sizeof(message), format);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_printf_or_die*(3) function.

**explain_message_errno_printf**

void explain_message_errno_printf(char *message, int message_size, int errnum, const char *format);

The **explain_message_errno_printf** function is used to obtain an explanation of an error returned by the *printf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*format* The original format, exactly as passed to the *printf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = printf(format);
if (result < 0 && errno != 0)
```

```
{
    int err = errno;
    char message[3000];
   explain_message_errno_printf(message, sizeof(message), err,
   format);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_printf_or_die*(3) function.

## SEE ALSO

*printf*(3) formatted output conversion

*explain_printf_or_die*(3)
formatted output conversion and report errors

## COPYRIGHT

## NAME

explain_printf_or_die − formatted output conversion and report errors

## SYNOPSIS

#include <libexplain/printf.h>

int explain_printf_or_die(const char *format);
int explain_printf_on_error(const char *format);

## DESCRIPTION

The **explain_printf_or_die** function is used to call the *printf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_printf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_printf_on_error** function is used to call the *printf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_printf*(3) function, but still returns to the caller.

*format* The format, exactly as to be passed to the *printf*(3) system call.

## RETURN VALUE

The **explain_printf_or_die** function only returns on success, see *printf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_printf_on_error** function always returns the value return by the wrapped *printf*(3) system call.

## EXAMPLE

The **explain_printf_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_printf_or_die(format);
```

## SEE ALSO

*printf*(3) formatted output conversion

*explain_printf*(3)
explain *printf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_program_name − manipulate the program name

## SYNOPSIS

```
#include <libexplain/libexplain.h>

const char *explain_program_name_get(void);
void explain_program_name_set(const char *name);
void explain_program_name_assemble(int yesno);
```

## DESCRIPTION

These functions may be used to manipulate libexplain's idea of the command name of the current process, and whether or not that name is included in error messages.

**explain_program_name_get**

const char *explain_program_name_get(void);

The **explain_program_name_get** function may be used to obtain the command name of the calling process. Depending on how capable `/proc` is on your system, or, failing that, how capable *lsof*(1) is on your system, this may or may not produce a sensible result. It works well on Linux.

Returns: pointer to string containing the command name (no slashes) of the calling process.

**explain_program_name_set**

void explain_program_name_set(const char *name);

The **explain_program_name_set** function may be used to set the libexplain libraries' idea of the command name of the calling process, setting the string to be returned by the *explain_program_name_get*(3) function. This overrides the automatic behavior, which can be quite desirable in commands that can be invoked with more than one name, *e.g.* if they are a hard link synonym.

This also sets the option to include the program name in all of the error messages issued by the *explain_*_or_die*(3) functions.

*name* The name of the calling process. Only the basename will be used if a path containing slashes is given.

**explain_program_name_assemble**

void explain_program_name_assemble(int yesno);

The explain_program_name_assemble function is used to control whether or not the name of the calling process is to be included in error messages issued by the *explain_*_or_die*(3) functions. If not explicitly set, is controlled by the EXPLAIN_OPTIONS environment variable, or defaults to true if not set there either.

*yesno* non-zero (true) to have program name included, zero (false) to have program name excluded.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_ptrace − explain *ptrace*(2) errors

## SYNOPSIS

#include <libexplain/ptrace.h>

const char *explain_ptrace(int request, pid_t pid, void *addr, void *data);
const char *explain_errno_ptrace(int errnum, int request, pid_t pid, void *addr, void *data);
void explain_message_ptrace(char *message, int message_size, int request, pid_t pid, void *addr, void *data);
void explain_message_errno_ptrace(char *message, int message_size, int errnum, int request, pid_t pid, void *addr, void *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *ptrace*(2) system call.

### explain_ptrace

const char *explain_ptrace(int request, pid_t pid, void *addr, void *data);

The **explain_ptrace** function is used to obtain an explanation of an error returned by the *ptrace*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*request* The original request, exactly as passed to the *ptrace*(2) system call.

*pid* The original pid, exactly as passed to the *ptrace*(2) system call.

*addr* The original addr, exactly as passed to the *ptrace*(2) system call.

*data* The original data, exactly as passed to the *ptrace*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = ptrace(request, pid, addr, data);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_ptrace(request, pid, addr,
   data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_ptrace_or_die*(3) function.

### explain_errno_ptrace

const char *explain_errno_ptrace(int errnum, int request, pid_t pid, void *addr, void *data);

The **explain_errno_ptrace** function is used to obtain an explanation of an error returned by the *ptrace*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*request* The original request, exactly as passed to the *ptrace*(2) system call.

*pid* The original pid, exactly as passed to the *ptrace*(2) system call.

*addr* The original addr, exactly as passed to the *ptrace*(2) system call.

*data* The original data, exactly as passed to the *ptrace*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = ptrace(request, pid, addr, data);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_ptrace(err, request,
   pid, addr, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_ptrace_or_die*(3) function.

**explain_message_ptrace**

void explain_message_ptrace(char *message, int message_size, int request, pid_t pid, void *addr, void *data);

The **explain_message_ptrace** function is used to obtain an explanation of an error returned by the *ptrace*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*request* The original request, exactly as passed to the *ptrace*(2) system call.

*pid* The original pid, exactly as passed to the *ptrace*(2) system call.

*addr* The original addr, exactly as passed to the *ptrace*(2) system call.

*data* The original data, exactly as passed to the *ptrace*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = ptrace(request, pid, addr, data);
if (result < 0)
{
    char message[3000];
   explain_message_ptrace(message, sizeof(message), request, pid,
   addr, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_ptrace_or_die*(3) function.

**explain_message_errno_ptrace**

void explain_message_errno_ptrace(char *message, int message_size, int errnum, int request, pid_t pid, void *addr, void *data);

The **explain_message_errno_ptrace** function is used to obtain an explanation of an error returned by the *ptrace*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*request* The original request, exactly as passed to the *ptrace*(2) system call.

*pid* The original pid, exactly as passed to the *ptrace*(2) system call.

*addr* The original addr, exactly as passed to the *ptrace*(2) system call.

*data* The original data, exactly as passed to the *ptrace*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = ptrace(request, pid, addr, data);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_ptrace(message, sizeof(message), err,
   request, pid, addr, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_ptrace_or_die*(3) function.

## SEE ALSO

*ptrace*(2)
process trace

*explain_ptrace_or_die*(3)
process trace and report errors

## COPYRIGHT

## NAME

explain_ptrace_or_die − process trace and report errors

## SYNOPSIS

#include <libexplain/ptrace.h>

long explain_ptrace_or_die(int request, pid_t pid, void *addr, void *data);
long explain_ptrace_on_error(int request, pid_t pid, void *addr, void *data);

## DESCRIPTION

The **explain_ptrace_or_die** function is used to call the *ptrace*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ptrace*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_ptrace_on_error** function is used to call the *ptrace*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ptrace*(3) function, but still returns to the caller.

*request* The request, exactly as to be passed to the *ptrace*(2) system call.

*pid* The pid, exactly as to be passed to the *ptrace*(2) system call.

*addr* The addr, exactly as to be passed to the *ptrace*(2) system call.

*data* The data, exactly as to be passed to the *ptrace*(2) system call.

## RETURN VALUE

The **explain_ptrace_or_die** function only returns on success, see *ptrace*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_ptrace_on_error** function always returns the value return by the wrapped *ptrace*(2) system call.

## EXAMPLE

The **explain_ptrace_or_die** function is intended to be used in a fashion similar to the following example:

```
long result = explain_ptrace_or_die(request, pid, addr, data);
```

## SEE ALSO

*ptrace*(2)
process trace

*explain_ptrace*(3)
explain *ptrace*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_putc − explain putc(3) errors

## SYNOPSIS

#include <libexplain/putc.h>

const char *explain_putc(int c, FILE *fp);
const char *explain_errno_putc(int errnum, int c, FILE *fp);
void explain_message_putc(char *message, int message_size, int c, FILE *fp);
void explain_message_errno_putc(char *message, int message_size, int errnum, int c, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *putc*(3) system call.

### explain_putc

const char *explain_putc(int c, FILE *fp);

The **explain_putc** function is used to obtain an explanation of an error returned by the *putc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (putc(c, fp) == EOF)
{
    fprintf(stderr, "%s\n", explain_putc(c, fp));
    exit(EXIT_FAILURE);
}
```

*c* The original c, exactly as passed to the *putc*(3) system call.

*fp* The original fp, exactly as passed to the *putc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_putc

const char *explain_errno_putc(int errnum, int c, FILE *fp);

The **explain_errno_putc** function is used to obtain an explanation of an error returned by the *putc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (putc(c, fp) == EOF)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_putc(err, c, fp));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*c* The original c, exactly as passed to the *putc*(3) system call.

*fp* The original fp, exactly as passed to the *putc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_putc**

void explain_message_putc(char *message, int message_size, int c, FILE *fp);

The **explain_message_putc** function may be used to obtain an explanation of an error returned by the *putc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (putc(c, fp) == EOF)
{
    char message[3000];
    explain_message_putc(message, sizeof(message), c, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*c* The original c, exactly as passed to the *putc*(3) system call.

*fp* The original fp, exactly as passed to the *putc*(3) system call.

**explain_message_errno_putc**

void explain_message_errno_putc(char *message, int message_size, int errnum, int c, FILE *fp);

The **explain_message_errno_putc** function may be used to obtain an explanation of an error returned by the *putc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (putc(c, fp) == EOF)
{
    int err = errno;
    char message[3000];
    explain_message_errno_putc(message, sizeof(message), err, c, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*c* The original c, exactly as passed to the *putc*(3) system call.

*fp* The original fp, exactly as passed to the *putc*(3) system call.

## SEE ALSO

*putc*(3) output of characters

*explain_putc_or_die*(3)
output of characters and report errors

## COPYRIGHT

## NAME

explain_putchar − explain putchar(3) errors

## SYNOPSIS

#include <libexplain/putchar.h>

const char *explain_putchar(int c);
const char *explain_errno_putchar(int errnum, int c);
void explain_message_putchar(char *message, int message_size, int c);
void explain_message_errno_putchar(char *message, int message_size, int errnum, int c);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *putchar*(3) system call.

### explain_putchar

const char *explain_putchar(int c);

The **explain_putchar** function is used to obtain an explanation of an error returned by the *putchar*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (putchar(c) == EOF)
{
    fprintf(stderr, "%s\n", explain_putchar(c));
    exit(EXIT_FAILURE);
}
```

*c* The original c, exactly as passed to the *putchar*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_putchar

const char *explain_errno_putchar(int errnum, int c);

The **explain_errno_putchar** function is used to obtain an explanation of an error returned by the *putchar*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (putchar(c) == EOF)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_putchar(err, c));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*c* The original c, exactly as passed to the *putchar*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_putchar**

void explain_message_putchar(char *message, int message_size, int c);

The **explain_message_putchar** function may be used to obtain an explanation of an error returned by the *putchar*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (putchar(c) == EOF)
{
    char message[3000];
    explain_message_putchar(message, sizeof(message), c);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*c* The original c, exactly as passed to the *putchar*(3) system call.

**explain_message_errno_putchar**

void explain_message_errno_putchar(char *message, int message_size, int errnum, int c);

The **explain_message_errno_putchar** function may be used to obtain an explanation of an error returned by the *putchar*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (putchar(c) == EOF)
{
    int err = errno;
    char message[3000];
    explain_message_errno_putchar(message, sizeof(message), err, c);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*c* The original c, exactly as passed to the *putchar*(3) system call.

## SEE ALSO

*putchar*(3)
output of characters

*explain_putchar_or_die*(3)
output of characters and report errors

## COPYRIGHT

## NAME

explain_putchar_or_die − output of characters and report errors

## SYNOPSIS

#include <libexplain/putchar.h>

void explain_putchar_or_die(int c);

## DESCRIPTION

The **explain_putchar_or_die** function is used to call the *putchar*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_putchar*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_putchar_or_die(c);
```

*c* The c, exactly as to be passed to the *putchar*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*putchar*(3)
output of characters

*explain_putchar*(3)
explain *putchar*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_putc_or_die − output of characters and report errors

## SYNOPSIS

#include <libexplain/putc.h>

void explain_putc_or_die(int c, FILE *fp);

## DESCRIPTION

The **explain_putc_or_die** function is used to call the *putc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_putc*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_putc_or_die(c, fp);
```

*c* The c, exactly as to be passed to the *putc*(3) system call.

*fp* The fp, exactly as to be passed to the *putc*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*putc*(3) output of characters

*explain_putc*(3)
explain *putc*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_putenv − explain *putenv*(3) errors

## SYNOPSIS

#include <libexplain/putenv.h>

const char *explain_putenv(char *string);
const char *explain_errno_putenv(int errnum, char *string);
void explain_message_putenv(char *message, int message_size, char *string);
void explain_message_errno_putenv(char *message, int message_size, int errnum, char *string);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *putenv*(3) system call.

### explain_putenv

const char *explain_putenv(char *string);

The **explain_putenv** function is used to obtain an explanation of an error returned by the *putenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*string* The original string, exactly as passed to the *putenv*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (putenv(string) < 0)
{
   fprintf(stderr, "%s\n", explain_putenv(string));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_putenv_or_die*(3) function.

### explain_errno_putenv

const char *explain_errno_putenv(int errnum, char *string);

The **explain_errno_putenv** function is used to obtain an explanation of an error returned by the *putenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*string* The original string, exactly as passed to the *putenv*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (putenv(string) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_putenv(err, string));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_putenv_or_die*(3) function.

**explain_message_putenv**

void explain_message_putenv(char *message, int message_size, char *string);

The **explain_message_putenv** function is used to obtain an explanation of an error returned by the *putenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*string* The original string, exactly as passed to the *putenv*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (putenv(string) < 0)
{
    char message[3000];
    explain_message_putenv(message, sizeof(message), string);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_putenv_or_die*(3) function.

**explain_message_errno_putenv**

void explain_message_errno_putenv(char *message, int message_size, int errnum, char *string);

The **explain_message_errno_putenv** function is used to obtain an explanation of an error returned by the *putenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*string* The original string, exactly as passed to the *putenv*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (putenv(string) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_putenv(message, sizeof(message), err,
    string);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_putenv_or_die*(3) function.

## SEE ALSO

*putenv*(3)
change or add an environment variable

*explain_putenv_or_die*(3)
change or add an environment variable and report errors

## COPYRIGHT

## NAME

explain_putenv_or_die − change or add an environment variable and report errors

## SYNOPSIS

#include <libexplain/putenv.h>

void explain_putenv_or_die(char *string);
int explain_putenv_on_error(char *string);

## DESCRIPTION

The **explain_putenv_or_die** function is used to call the *putenv*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_putenv*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_putenv_on_error** function is used to call the *putenv*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_putenv*(3) function, but still returns to the caller.

*string* The string, exactly as to be passed to the *putenv*(3) system call.

## RETURN VALUE

The **explain_putenv_or_die** function only returns on success, see *putenv*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_putenv_on_error** function always returns the value return by the wrapped *putenv*(3) system call.

## EXAMPLE

The **explain_putenv_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_putenv_or_die(string);
```

## SEE ALSO

*putenv*(3)
change or add an environment variable

*explain_putenv*(3)
explain *putenv*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_puts − explain *puts*(3) errors

## SYNOPSIS

#include <libexplain/puts.h>

const char *explain_puts(const char *s);
const char *explain_errno_puts(int errnum, const char *s);
void explain_message_puts(char *message, int message_size, const char *s);
void explain_message_errno_puts(char *message, int message_size, int errnum, const char *s);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *puts*(3) system call.

### explain_puts

const char *explain_puts(const char *s);

The **explain_puts** function is used to obtain an explanation of an error returned by the *puts*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*s* The original s, exactly as passed to the *puts*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (puts(s) < 0)
{
   fprintf(stderr, "%s\n", explain_puts(s));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_puts_or_die*(3) function.

### explain_errno_puts

const char *explain_errno_puts(int errnum, const char *s);

The **explain_errno_puts** function is used to obtain an explanation of an error returned by the *puts*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*s* The original s, exactly as passed to the *puts*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (puts(s) < 0)
{
```

```
        int err = errno;
       fprintf(stderr, "%s\n", explain_errno_puts(err, s));
        exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_puts_or_die*(3) function.

**explain_message_puts**

void explain_message_puts(char *message, int message_size, const char *s);

The **explain_message_puts** function is used to obtain an explanation of an error returned by the *puts*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*s* The original s, exactly as passed to the *puts*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (puts(s) < 0)
{
    char message[3000];
   explain_message_puts(message, sizeof(message), s);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_puts_or_die*(3) function.

**explain_message_errno_puts**

void explain_message_errno_puts(char *message, int message_size, int errnum, const char *s);

The **explain_message_errno_puts** function is used to obtain an explanation of an error returned by the *puts*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*s* The original s, exactly as passed to the *puts*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (puts(s) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_puts(message, sizeof(message), err, s);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_puts_or_die*(3) function.

## SEE ALSO

*puts*(3) write a string and a trailing newline to stdout

*explain_puts_or_die*(3)
write a string and a trailing newline to stdout and report errors

## COPYRIGHT

## NAME

explain_puts_or_die − write a string and a newline to stdout and report errors

## SYNOPSIS

#include <libexplain/puts.h>

void explain_puts_or_die(const char *s);
int explain_puts_on_error(const char *s);

## DESCRIPTION

The **explain_puts_or_die** function is used to call the *puts*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_puts*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_puts_on_error** function is used to call the *puts*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_puts*(3) function, but still returns to the caller.

*s* The s, exactly as to be passed to the *puts*(3) system call.

## RETURN VALUE

The **explain_puts_or_die** function only returns on success, see *puts*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_puts_on_error** function always returns the value return by the wrapped *puts*(3) system call.

## EXAMPLE

The **explain_puts_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_puts_or_die(s);
```

## SEE ALSO

*puts*(3) write a string and a trailing newline to stdout

*explain_puts*(3)
explain *puts*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_putw − explain *putw*(3) errors

## SYNOPSIS

#include <libexplain/putw.h>

const char *explain_putw(int value, FILE *fp);
const char *explain_errno_putw(int errnum, int value, FILE *fp);
void explain_message_putw(char *message, int message_size, int value, FILE *fp);
void explain_message_errno_putw(char *message, int message_size, int errnum, int value, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *putw*(3) system call.

### explain_putw

const char *explain_putw(int value, FILE *fp);

The **explain_putw** function is used to obtain an explanation of an error returned by the *putw*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*value* The original value, exactly as passed to the *putw*(3) system call.

*fp* The original fp, exactly as passed to the *putw*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (putw(value, fp) < 0)
{
   fprintf(stderr, "%s\n", explain_putw(value, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_putw_or_die*(3) function.

### explain_errno_putw

const char *explain_errno_putw(int errnum, int value, FILE *fp);

The **explain_errno_putw** function is used to obtain an explanation of an error returned by the *putw*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*value* The original value, exactly as passed to the *putw*(3) system call.

*fp* The original fp, exactly as passed to the *putw*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (putw(value, fp) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_putw(err, value, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_putw_or_die*(3) function.

**explain_message_putw**

void explain_message_putw(char *message, int message_size, int value, FILE *fp);

The **explain_message_putw** function is used to obtain an explanation of an error returned by the *putw*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*value* The original value, exactly as passed to the *putw*(3) system call.

*fp* The original fp, exactly as passed to the *putw*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (putw(value, fp) < 0)
{
    char message[3000];
    explain_message_putw(message, sizeof(message), value, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_putw_or_die*(3) function.

**explain_message_errno_putw**

void explain_message_errno_putw(char *message, int message_size, int errnum, int value, FILE *fp);

The **explain_message_errno_putw** function is used to obtain an explanation of an error returned by the *putw*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*value* The original value, exactly as passed to the *putw*(3) system call.

*fp* The original fp, exactly as passed to the *putw*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (putw(value, fp) < 0)
{
    int err = errno;
    char message[3000];
```

```
        explain_message_errno_putw(message, sizeof(message), err,
        value, fp);
         fprintf(stderr, "%s\n", message);
         exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_putw_or_die*(3) function.

## SEE ALSO

*putw*(3) output a word (int)

*explain_putw_or_die*(3)
output a word (int) and report errors

## COPYRIGHT

## NAME

explain_putw_or_die − output a word (int) and report errors

## SYNOPSIS

#include <libexplain/putw.h>

void explain_putw_or_die(int value, FILE *fp);
int explain_putw_on_error(int value, FILE *fp);

## DESCRIPTION

The **explain_putw_or_die** function is used to call the *putw*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_putw*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_putw_on_error** function is used to call the *putw*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_putw*(3) function, but still returns to the caller.

*value* The value, exactly as to be passed to the *putw*(3) system call.

*fp* The fp, exactly as to be passed to the *putw*(3) system call.

## RETURN VALUE

The **explain_putw_or_die** function only returns on success, see *putw*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_putw_on_error** function always returns the value return by the wrapped *putw*(3) system call.

## EXAMPLE

The **explain_putw_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_putw_or_die(value, fp);
```

## SEE ALSO

*putw*(3) output a word (int)

*explain_putw*(3)
explain *putw*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_pwrite − explain pwrite(2) errors

## SYNOPSIS

#include <libexplain/pwrite.h>

const char *explain_pwrite(int fildes, const void *data, size_t data_size, off_t offset);
const char *explain_errno_pwrite(int errnum, int fildes, const void *data, size_t data_size, off_t offset);
void explain_message_pwrite(char *message, int message_size, int fildes, const void *data, size_t data_size, off_t offset);
void explain_message_errno_pwrite(char *message, int message_size, int errnum, int fildes, const void *data, size_t data_size, off_t offset);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *pwrite*(2) system call.

### explain_pwrite

const char *explain_pwrite(int fildes, const void *data, size_t data_size, off_t offset);

The **explain_pwrite** function is used to obtain an explanation of an error returned by the *pwrite*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *pwrite*(2) system call.

*data* The original data, exactly as passed to the *pwrite*(2) system call.

*data_size*
The original data_size, exactly as passed to the *pwrite*(2) system call.

*offset* The original offset, exactly as passed to the *pwrite*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = pwrite(fildes, data, data_size, offset);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_pwrite(fildes, data,
   data_size, offset));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pwrite_or_die*(3) function.

### explain_errno_pwrite

const char *explain_errno_pwrite(int errnum, int fildes, const void *data, size_t data_size, off_t offset);

The **explain_errno_pwrite** function is used to obtain an explanation of an error returned by the *pwrite*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *pwrite*(2) system call.

*data* The original data, exactly as passed to the *pwrite*(2) system call.

*data_size*
The original data_size, exactly as passed to the *pwrite*(2) system call.

*offset* The original offset, exactly as passed to the *pwrite*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = pwrite(fildes, data, data_size, offset);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_pwrite(err, fildes,
   data, data_size, offset));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pwrite_or_die*(3) function.

**explain_message_pwrite**

void explain_message_pwrite(char *message, int message_size, int fildes, const void *data, size_t data_size, off_t offset);

The **explain_message_pwrite** function is used to obtain an explanation of an error returned by the *pwrite*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *pwrite*(2) system call.

*data* The original data, exactly as passed to the *pwrite*(2) system call.

*data_size*
The original data_size, exactly as passed to the *pwrite*(2) system call.

*offset* The original offset, exactly as passed to the *pwrite*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = pwrite(fildes, data, data_size, offset);
if (result < 0)
{
    char message[3000];
   explain_message_pwrite(message, sizeof(message), fildes, data,
   data_size, offset);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pwrite_or_die*(3) function.

**explain_message_errno_pwrite**

void explain_message_errno_pwrite(char *message, int message_size, int errnum, int fildes, const void *data, size_t data_size, off_t offset);

The **explain_message_errno_pwrite** function is used to obtain an explanation of an error returned by the *pwrite*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *pwrite*(2) system call.

*data* The original data, exactly as passed to the *pwrite*(2) system call.

*data_size*
The original data_size, exactly as passed to the *pwrite*(2) system call.

*offset* The original offset, exactly as passed to the *pwrite*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = pwrite(fildes, data, data_size, offset);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_pwrite(message, sizeof(message), err,
   fildes, data, data_size, offset);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_pwrite_or_die*(3) function.

## SEE ALSO

*pwrite*(2)
read from or write to a file descriptor at a given offset

*explain_pwrite_or_die*(3)
read from or write to a file descriptor at a given offset and report errors

## COPYRIGHT

## NAME

explain_pwrite_or_die − seek and write to a file descriptor and report errors

## SYNOPSIS

#include <libexplain/pwrite.h>

ssize_t explain_pwrite_or_die(int fildes, const void *data, size_t data_size, off_t offset);
ssize_t explain_pwrite_on_error(int fildes, const void *data, size_t data_size, off_t offset))

## DESCRIPTION

The **explain_pwrite_or_die** function is used to call the *pwrite*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_pwrite*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_pwrite_on_error** function is used to call the *pwrite*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_pwrite*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *pwrite*(2) system call.

*data* The data, exactly as to be passed to the *pwrite*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *pwrite*(2) system call.

*offset* The offset, exactly as to be passed to the *pwrite*(2) system call.

## RETURN VALUE

The **explain_pwrite_or_die** function only returns on success, see *pwrite*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_pwrite_on_error** function always returns the value return by the wrapped *pwrite*(2) system call.

## EXAMPLE

The **explain_pwrite_or_die** function is intended to be used in a fashion similar to the following example:

```
ssize_t result = explain_pwrite_or_die(fildes, data, data_size, offset);
```

## SEE ALSO

*pwrite*(2)
read from or write to a file descriptor at a given offset

*explain_pwrite*(3)
explain *pwrite*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_raise − explain *raise*(3) errors

## SYNOPSIS

#include <libexplain/raise.h>

const char *explain_raise(int sig);
const char *explain_errno_raise(int errnum, int sig);
void explain_message_raise(char *message, int message_size, int sig);
void explain_message_errno_raise(char *message, int message_size, int errnum, int sig);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *raise*(3) system call.

### explain_raise

const char *explain_raise(int sig);

The **explain_raise** function is used to obtain an explanation of an error returned by the *raise*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*sig* The original sig, exactly as passed to the *raise*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (raise(sig) < 0)
{
   fprintf(stderr, "%s\n", explain_raise(sig));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_raise_or_die*(3) function.

### explain_errno_raise

const char *explain_errno_raise(int errnum, int sig);

The **explain_errno_raise** function is used to obtain an explanation of an error returned by the *raise*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*sig* The original sig, exactly as passed to the *raise*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (raise(sig) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_raise(err, sig));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_raise_or_die*(3) function.

**explain_message_raise**

void explain_message_raise(char *message, int message_size, int sig);

The **explain_message_raise** function is used to obtain an explanation of an error returned by the *raise*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*sig* The original sig, exactly as passed to the *raise*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (raise(sig) < 0)
{
    char message[3000];
    explain_message_raise(message, sizeof(message), sig);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_raise_or_die*(3) function.

**explain_message_errno_raise**

void explain_message_errno_raise(char *message, int message_size, int errnum, int sig);

The **explain_message_errno_raise** function is used to obtain an explanation of an error returned by the *raise*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*sig* The original sig, exactly as passed to the *raise*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (raise(sig) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_raise(message, sizeof(message), err,
    sig);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_raise_or_die*(3) function.

## SEE ALSO

*raise*(3) send a signal to the caller

*explain_raise_or_die*(3)
send a signal to the caller and report errors

## COPYRIGHT

## NAME

explain_raise_or_die − send a signal to the caller and report errors

## SYNOPSIS

#include <libexplain/raise.h>

void explain_raise_or_die(int sig);
int explain_raise_on_error(int sig);

## DESCRIPTION

The **explain_raise_or_die** function is used to call the *raise*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_raise*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_raise_on_error** function is used to call the *raise*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_raise*(3) function, but still returns to the caller.

*sig* The sig, exactly as to be passed to the *raise*(3) system call.

## RETURN VALUE

The **explain_raise_or_die** function only returns on success, see *raise*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_raise_on_error** function always returns the value return by the wrapped *raise*(3) system call.

## EXAMPLE

The **explain_raise_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_raise_or_die(sig);
```

## SEE ALSO

*raise*(3) send a signal to the caller

*explain_raise*(3)
explain *raise*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_read − explain read(2) errors

## SYNOPSIS

#include <libexplain/read.h>
const char *explain_read(int fildes, const void *data, long data_size);
const char *explain_errno_read(int errnum, int fildes, const void *data, long data_size);
void explain_message_read(char *message, int message_size, int fildes, const void *data, long data_size);
void explain_message_errno_read(char *message, int message_size, int errnum, int fildes, const void *data, long data_size);

## DESCRIPTION

These functions may be used to obtain an explanation for *read*(2) errors.

### explain_read

const char *explain_read(int fildes, const void *data, long data_size);

The explain_read function may be used to obtain a human readable explanation of what went wrong in a *read*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The error number will be picked up from the *errno* global variable.

This function is intended to be used in a fashion similar to the following example:

```
ssize_t n = read(fd, data, data_size);
if (n < 0)
{
    fprintf(stderr, "%s\n", explain_read(fd, data, data_size));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *read*(2) system call.

*data* The original data, exactly as passed to the *read*(2) system call.

*data_size*
The original data_size, exactly as passed to the *read*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_read

const char *explain_errno_read(int errnum, int fildes, const void *data, long data_size);

The explain_errno_read function may be used to obtain a human readable explanation of what went wrong in a *read*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
ssize_t n = read(fd, data, data_size);
if (n < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_read(err, fd, data, data_size));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtain from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *read*(2) system call.

*data* The original data, exactly as passed to the *read*(2) system call.

*data_size*
The original data_size, exactly as passed to the *read*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_read**

void explain_message_read(char *message, int message_size, int fildes, const void *data, long data_size);

The explain_message_read function may be used to obtain a human readable explanation of what went wrong in a *read*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The error number will be picked up from the *errno* global variable.

This function is intended to be used in a fashion similar to the following example:

```
ssize_t n = read(fd, data, data_size);
if (n < 0)
{
    char message[3000];
    explain_message_read(message, sizeof(message), fd, data, data_size));
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *read*(2) system call.

*data* The original data, exactly as passed to the *read*(2) system call.

*data_size*
The original data_size, exactly as passed to the *read*(2) system call.

**Note:** Given a suitably thread safe buffer, this function is thread safe.

**explain_message_errno_read**

void explain_message_errno_read(char *message, int message_size, int errnum, int fildes, const void *data, long data_size);

The explain_message_errno_read function may be used to obtain a human readable explanation of what went wrong in a *read*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
ssize_t n = read(fd, data, data_size);
if (n < 0)
{
```

```
        int err = errno;
        char message[3000];
        explain_message_errno_read(message, sizeof(message), err,
            fd, data, data_size);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtain from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *read*(2) system call.

*data* The original data, exactly as passed to the *read*(2) system call.

*data_size*
The original data_size, exactly as passed to the *read*(2) system call.

**Note:** Given a suitably thread safe buffer, this function is thread safe.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_readdir − explain readdir(2) errors

## SYNOPSIS

#include <libexplain/readdir.h>

const char *explain_readdir(DIR *dir);
const char *explain_errno_readdir(int errnum, DIR *dir);
void explain_message_readdir(char *message, int message_size, DIR *dir);
void explain_message_errno_readdir(char *message, int message_size, int errnum, DIR *dir);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *readdir*(2) system call.

### explain_readdir

const char *explain_readdir(DIR *dir);

The **explain_readdir** function is used to obtain an explanation of an error returned by the *readdir*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
struct dirent *dep = readdir(dir);
if (!dep && errno != 0)
{
    fprintf(stderr, "%s\n", explain_readdir(dir));
    exit(EXIT_FAILURE);
}
```

*dir* The original dir, exactly as passed to the *readdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_readdir

const char *explain_errno_readdir(int errnum, DIR *dir);

The **explain_errno_readdir** function is used to obtain an explanation of an error returned by the *readdir*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
struct dirent *dep = readdir(dir);
int err = errno;
if (!dep && errno != 0)
{
    fprintf(stderr, "%s\n", explain_errno_readdir(err, dir));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *readdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_readdir**

void explain_message_readdir(char *message, int message_size, DIR *dir);

The **explain_message_readdir** function may be used to obtain an explanation of an error returned by the *readdir*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
struct dirent *dep = readdir(dir);
if (!dep && errno != 0)
{
    char message[3000];
    explain_message_readdir(message, sizeof(message), dir);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*dir* The original dir, exactly as passed to the *readdir*(2) system call.

**explain_message_errno_readdir**

void explain_message_errno_readdir(char *message, int message_size, int errnum, DIR *dir);

The **explain_message_errno_readdir** function may be used to obtain an explanation of an error returned by the *readdir*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
struct dirent *dep = readdir(dir);
int err = errno;
if (!dep && errno != 0)
{
    char message[3000];
    explain_message_errno_readdir(message, sizeof(message), err, dir);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *readdir*(2) system call.

## SEE ALSO

*readdir*(2)
read directory entry

*explain_readdir_or_die*(3)
read directory entry and report errors

## COPYRIGHT

## NAME

explain_readdir_or_die − read directory entry and report errors

## SYNOPSIS

#include <libexplain/readdir.h>

struct dirent *explain_readdir_or_die(DIR *dir);

## DESCRIPTION

The **explain_readdir_or_die** function is used to call the *readdir*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_readdir*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_readdir_or_die(dir);
```

*dir* The dir, exactly as to be passed to the *readdir*(2) system call.

Returns: a pointer to a dirent structure, or NULL if end-of-file is reached. On failure, prints an explanation and exits.

## SEE ALSO

*readdir*(2)
read directory entry

*explain_readdir*(3)
explain *readdir*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_readlink − explain readlink(2) errors

## SYNOPSIS

#include <libexplain/readlink.h>

const char *explain_readlink(const char *pathname, char *data, size_t data_size);
const char *explain_errno_readlink(int errnum, const char *pathname, char *data, size_t data_size);
void explain_message_readlink(char *message, int message_size, const char *pathname, char *data, size_t data_size);
void explain_message_errno_readlink(char *message, int message_size, int errnum, const char *pathname, char *data, size_t data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *readlink*(2) system call.

### explain_readlink

const char *explain_readlink(const char *pathname, char *data, size_t data_size);

The **explain_readlink** function is used to obtain an explanation of an error returned by the *readlink*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (readlink(pathname, data, data_size) < 0)
{
    fprintf(stderr, "%s\n", explain_readlink(pathname, data, data_size));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *readlink*(2) system call.

*data*
: The original data, exactly as passed to the *readlink*(2) system call.

*data_size*
: The original data_size, exactly as passed to the *readlink*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_readlink

const char *explain_errno_readlink(int errnum, const char *pathname, char *data, size_t data_size);

The **explain_errno_readlink** function is used to obtain an explanation of an error returned by the *readlink*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (readlink(pathname, data, data_size) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_readlink(err, pathname, data,
        data_size));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *readlink*(2) system call.

*data* The original data, exactly as passed to the *readlink*(2) system call.

*data_size*
The original data_size, exactly as passed to the *readlink*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_readlink**

void explain_message_readlink(char *message, int message_size, const char *pathname, char *data, size_t data_size);

The **explain_message_readlink** function may be used to obtain an explanation of an error returned by the *readlink*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (readlink(pathname, data, data_size) < 0)
{
    char message[3000];
    explain_message_readlink(message, sizeof(message), pathname, data,
        data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *readlink*(2) system call.

*data* The original data, exactly as passed to the *readlink*(2) system call.

*data_size*
The original data_size, exactly as passed to the *readlink*(2) system call.

**explain_message_errno_readlink**

void explain_message_errno_readlink(char *message, int message_size, int errnum, const char *pathname, char *data, size_t data_size);

The **explain_message_errno_readlink** function may be used to obtain an explanation of an error returned by the *readlink*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (readlink(pathname, data, data_size) < 0)
{
    int err = errno;
```

```
        char message[3000];
        explain_message_errno_readlink(message, sizeof(message), err, pathname,
            data, data_size);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *readlink*(2) system call.

*data* The original data, exactly as passed to the *readlink*(2) system call.

*data_size*
The original data_size, exactly as passed to the *readlink*(2) system call.

## SEE ALSO

*readlink*(2)
blah blah blah

*explain_readlink_or_die*(3)
blah blah blah and report errors

## COPYRIGHT

## NAME

explain_readlink_or_die − read value of a symbolic link and report errors

## SYNOPSIS

#include <libexplain/readlink.h>

ssize_t explain_readlink_or_die(const char *pathname, char *data, size_t data_size);
ssize_t explain_readlink_on_error(const char *pathname, char *data, size_t data_size))

## DESCRIPTION

The **explain_readlink_or_die** function is used to call the *readlink*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_readlink*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_readlink_on_error** function is used to call the *readlink*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_readlink*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *readlink*(2) system call.

*data*
: The data, exactly as to be passed to the *readlink*(2) system call.

*data_size*
: The data_size, exactly as to be passed to the *readlink*(2) system call.

## RETURN VALUE

The **explain_readlink_or_die** function only returns on success, see *readlink*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_readlink_on_error** function always returns the value return by the wrapped *readlink*(2) system call.

## EXAMPLE

The **explain_readlink_or_die** function is intended to be used in a fashion similar to the following example:

```
ssize_t result = explain_readlink_or_die(pathname, data, data_size);
```

## SEE ALSO

*readlink*(2)
: read value of a symbolic link

*explain_readlink*(3)
: explain *readlink*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_read_or_die − read from a file descriptor and report errors

## SYNOPSIS

#include <libexplain/read.h>

long explain_read_or_die(int fildes, const void *data, long data_size);

## DESCRIPTION

The **explain_read_or_die** function is used to call the *read*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_read*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_read_or_die(fildes, data, data_size);
```

*fildes* The fildes, exactly as to be passed to the *read*(2) system call.

*data* The data, exactly as to be passed to the *read*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *read*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*read*(2) read from a file descriptor

*explain_read*(3)
explain *read*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_readv − explain readv(2) errors

## SYNOPSIS

#include <libexplain/readv.h>

const char *explain_readv(int fildes, const struct iovec *iov, int iovcnt);
const char *explain_errno_readv(int errnum, int fildes, const struct iovec *iov, int iovcnt);
void explain_message_readv(char *message, int message_size, int fildes, const struct iovec *iov, int iovcnt);
void explain_message_errno_readv(char *message, int message_size, int errnum, int fildes, const struct iovec *iov, int iovcnt);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *readv*(2) system call.

### explain_readv

const char *explain_readv(int fildes, const struct iovec *iov, int iovcnt);

The **explain_readv** function is used to obtain an explanation of an error returned by the *readv*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *readv*(2) system call.

*iov* The original iov, exactly as passed to the *readv*(2) system call.

*iovcnt* The original iovcnt, exactly as passed to the *readv*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = readv(fildes, iov, iovcnt);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_readv(fildes, iov, iovcnt));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_readv_or_die*(3) function.

### explain_errno_readv

const char *explain_errno_readv(int errnum, int fildes, const struct iovec *iov, int iovcnt);

The **explain_errno_readv** function is used to obtain an explanation of an error returned by the *readv*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *readv*(2) system call.

*iov* The original iov, exactly as passed to the *readv*(2) system call.

*iovcnt* The original iovcnt, exactly as passed to the *readv*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = readv(fildes, iov, iovcnt);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_readv(err, fildes, iov,
   iovcnt));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_readv_or_die*(3) function.

**explain_message_readv**

void explain_message_readv(char *message, int message_size, int fildes, const struct iovec *iov, int iovcnt);

The **explain_message_readv** function is used to obtain an explanation of an error returned by the *readv*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *readv*(2) system call.

*iov* The original iov, exactly as passed to the *readv*(2) system call.

*iovcnt* The original iovcnt, exactly as passed to the *readv*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = readv(fildes, iov, iovcnt);
if (result < 0)
{
    char message[3000];
   explain_message_readv(message, sizeof(message), fildes, iov,
   iovcnt);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_readv_or_die*(3) function.

**explain_message_errno_readv**

void explain_message_errno_readv(char *message, int message_size, int errnum, int fildes, const struct iovec *iov, int iovcnt);

The **explain_message_errno_readv** function is used to obtain an explanation of an error returned by the *readv*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *readv*(2) system call.

*iov* The original iov, exactly as passed to the *readv*(2) system call.

*iovcnt* The original iovcnt, exactly as passed to the *readv*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = readv(fildes, iov, iovcnt);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_readv(message, sizeof(message), err,
   fildes, iov, iovcnt);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_readv_or_die*(3) function.

## SEE ALSO

*readv*(2) read data into multiple buffers

*explain_readv_or_die*(3)
read data into multiple buffers and report errors

## COPYRIGHT

## NAME

explain_readv_or_die − read data into multiple buffers and report errors

## SYNOPSIS

#include <libexplain/readv.h>

ssize_t explain_readv_or_die(int fildes, const struct iovec *iov, int iovcnt);
ssize_t explain_readv_on_error(int fildes, const struct iovec *iov, int iovcnt);

## DESCRIPTION

The **explain_readv_or_die** function is used to call the *readv*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_readv*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_readv_on_error** function is used to call the *readv*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_readv*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *readv*(2) system call.

*iov* The iov, exactly as to be passed to the *readv*(2) system call.

*iovcnt* The iovcnt, exactly as to be passed to the *readv*(2) system call.

## RETURN VALUE

The **explain_readv_or_die** function only returns on success, see *readv*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_readv_on_error** function always returns the value return by the wrapped *readv*(2) system call.

## EXAMPLE

The **explain_readv_or_die** function is intended to be used in a fashion similar to the following example:

```
ssize_t result = explain_readv_or_die(fildes, iov, iovcnt);
```

## SEE ALSO

*readv*(2) read data into multiple buffers

*explain_readv*(3)
explain *readv*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_realloc − explain realloc(3) errors

## SYNOPSIS

#include <libexplain/realloc.h>

const char *explain_realloc(void *ptr, size_t size);
const char *explain_errno_realloc(int errnum, void *ptr, size_t size);
void explain_message_realloc(char *message, int message_size, void *ptr, size_t size);
void explain_message_errno_realloc(char *message, int message_size, int errnum, void *ptr, size_t size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *realloc*(3) system call.

### explain_realloc

const char *explain_realloc(void *ptr, size_t size);

The **explain_realloc** function is used to obtain an explanation of an error returned by the *realloc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
void *new_ptr = realloc(ptr, size);
if (!new_ptr)
{
    fprintf(stderr, "%s\n", explain_realloc(ptr, size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_realloc_or_die*(3) function.

*ptr* The original ptr, exactly as passed to the *realloc*(3) system call.

*size* The original size, exactly as passed to the *realloc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_realloc

const char *explain_errno_realloc(int errnum, void *ptr, size_t size);

The **explain_errno_realloc** function is used to obtain an explanation of an error returned by the *realloc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
void *new_ptr = realloc(ptr, size);
if (!new_ptr)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_realloc(err, ptr, size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_realloc_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*ptr* The original ptr, exactly as passed to the *realloc*(3) system call.

*size* The original size, exactly as passed to the *realloc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_realloc**

void explain_message_realloc(char *message, int message_size, void *ptr, size_t size);

The **explain_message_realloc** function may be used to obtain an explanation of an error returned by the *realloc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
void *new_ptr = realloc(ptr, size);
if (!new_ptr)
{
    char message[3000];
    explain_message_realloc(message, sizeof(message), ptr, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_realloc_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*ptr* The original ptr, exactly as passed to the *realloc*(3) system call.

*size* The original size, exactly as passed to the *realloc*(3) system call.

**explain_message_errno_realloc**

void explain_message_errno_realloc(char *message, int message_size, int errnum, void *ptr, size_t size);

The **explain_message_errno_realloc** function may be used to obtain an explanation of an error returned by the *realloc*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
void *new_ptr = realloc(ptr, size);
if (!new_ptr)
{
    int err = errno;
    char message[3000];
    explain_message_errno_realloc(message, sizeof(message), err, ptr, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_realloc_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ptr* The original ptr, exactly as passed to the *realloc*(3) system call.

*size* The original size, exactly as passed to the *realloc*(3) system call.

## SEE ALSO

*realloc*(3)
Allocate and free dynamic memory

*explain_realloc_or_die*(3)
Allocate and free dynamic memory and report errors

## COPYRIGHT

## NAME

explain_realloc_or_die − Allocate and free dynamic memory and report errors

## SYNOPSIS

#include <libexplain/realloc.h>

void explain_realloc_or_die(void *ptr, size_t size);

## DESCRIPTION

The **explain_realloc_or_die** function is used to call the *realloc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_realloc*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
void *new_ptr = explain_realloc_or_die(ptr, size);
```

*ptr* The ptr, exactly as to be passed to the *realloc*(3) system call.

*size* The size, exactly as to be passed to the *realloc*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*realloc*(3)
Allocate and free dynamic memory

*explain_realloc*(3)
explain *realloc*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_realpath − explain *realpath*(3) errors

## SYNOPSIS

#include <libexplain/realpath.h>

const char *explain_realpath(const char *pathname, char *resolved_pathname);
const char *explain_errno_realpath(int errnum, const char *pathname, char *resolved_pathname);
void explain_message_realpath(char *message, int message_size, const char *pathname, char *resolved_pathname);
void explain_message_errno_realpath(char *message, int message_size, int errnum, const char *pathname, char *resolved_pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *realpath*(3) system call.

### explain_realpath

const char *explain_realpath(const char *pathname, char *resolved_pathname);

The **explain_realpath** function is used to obtain an explanation of an error returned by the *realpath*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *realpath*(3) system call.

*resolved_pathname*
: The original resolved_pathname, exactly as passed to the *realpath*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = realpath(pathname, resolved_pathname);
if (!result)
{
   fprintf(stderr, "%s\n", explain_realpath(pathname,
   resolved_pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_realpath_or_die*(3) function.

### explain_errno_realpath

const char *explain_errno_realpath(int errnum, const char *pathname, char *resolved_pathname);

The **explain_errno_realpath** function is used to obtain an explanation of an error returned by the *realpath*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *realpath*(3) system call.

*resolved_pathname*
The original resolved_pathname, exactly as passed to the *realpath*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = realpath(pathname, resolved_pathname);
if (!result)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_realpath(err, pathname,
   resolved_pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_realpath_or_die*(3) function.

**explain_message_realpath**

void explain_message_realpath(char *message, int message_size, const char *pathname, char *resolved_pathname);

The **explain_message_realpath** function is used to obtain an explanation of an error returned by the *realpath*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *realpath*(3) system call.

*resolved_pathname*
The original resolved_pathname, exactly as passed to the *realpath*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = realpath(pathname, resolved_pathname);
if (!result)
{
    char message[3000];
   explain_message_realpath(message, sizeof(message), pathname,
   resolved_pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_realpath_or_die*(3) function.

**explain_message_errno_realpath**

void explain_message_errno_realpath(char *message, int message_size, int errnum, const char *pathname, char *resolved_pathname);

The **explain_message_errno_realpath** function is used to obtain an explanation of an error returned by the *realpath*(3) system call. The least the message will contain is the value of `strerror(errno)`, but

usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *realpath*(3) system call.

*resolved_pathname*
The original resolved_pathname, exactly as passed to the *realpath*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = realpath(pathname, resolved_pathname);
if (!result)
{
    int err = errno;
    char message[3000];
   explain_message_errno_realpath(message, sizeof(message), err,
   pathname, resolved_pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_realpath_or_die*(3) function.

## SEE ALSO

*realpath*(3)
return the canonicalized absolute pathname

*explain_realpath_or_die*(3)
return the canonicalized absolute pathname and report errors

## COPYRIGHT

## NAME

explain_realpath_or_die − return canonical pathname and report errors

## SYNOPSIS

#include <libexplain/realpath.h>

char *explain_realpath_or_die(const char *pathname, char *resolved_pathname);
char *explain_realpath_on_error(const char *pathname, char *resolved_pathname);

## DESCRIPTION

The **explain_realpath_or_die** function is used to call the *realpath*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_realpath*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_realpath_on_error** function is used to call the *realpath*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_realpath*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *realpath*(3) system call.

*resolved_pathname*
: The resolved_pathname, exactly as to be passed to the *realpath*(3) system call.

## RETURN VALUE

The **explain_realpath_or_die** function only returns on success, see *realpath*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_realpath_on_error** function always returns the value return by the wrapped *realpath*(3) system call.

## EXAMPLE

The **explain_realpath_or_die** function is intended to be used in a fashion similar to the following example:

```
char *result = explain_realpath_or_die(pathname, resolved_pathname);
```

## SEE ALSO

*realpath*(3)
: return the canonicalized absolute pathname

*explain_realpath*(3)
: explain *realpath*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_remove − explain remove(2) errors

## SYNOPSIS

#include <libexplain/remove.h>

const char *explain_remove(const char *pathname);
const char *explain_errno_remove(int errnum, const char *pathname);
void explain_message_remove(char *message, int message_size, const char *pathname);
void explain_message_errno_remove(char *message, int message_size, int errnum, const char *pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *remove*(2) system call.

### explain_remove

const char *explain_remove(const char *pathname);

The **explain_remove** function may be used to describe errors returned by the *remove*() system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (remove(pathname) < 0)
{
    fprintf(stderr, "%s\n", explain_remove(pathname));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *remove*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_remove

const char *explain_errno_remove(int errnum, const char *pathname);

The **explain_errno_remove** function may be used to describe errors returned by the *remove*() system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (remove(pathname) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_remove(err, pathname));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *remove*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_removevoid explain_message_remove(char *message,**
int message_size, const char *pathname);

The **explain_message_remove** function may be used to describe errors returned by the *remove*() system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (remove(pathname) < 0)
{
    char message[3000];
    explain_message_remove(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *remove*(2) system call.

**explain_message_errno_remove**
void explain_message_errno_remove(char *message, int message_size, int errnum, const char *pathname);

The **explain_message_errno_remove** function may be used to describe errors returned by the *remove*() system call. The least the message will contain is the value of *strerror(errnum)*, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (remove(pathname) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_remove(message, sizeof(message), err, pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *remove*(2) system call.

## SEE ALSO

*remove* delete a name and possibly the file it refers to

*explain_remove_or_die*
delete a file and report errors

## COPYRIGHT

## NAME

explain_remove_or_die − delete a file and report errors

## SYNOPSIS

#include <libexplain/remove.h>

void explain_remove_or_die(const char *pathname);

## DESCRIPTION

The **explain_remove_or_die** function is used to call the *remove*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_remove*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_remove_or_die(pathname);
```

*pathname*
: The pathname, exactly as to be passed to the *remove*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*remove*(2)
: delete a name and possibly the file it refers to

*explain_remove*(3)
: explain *remove*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_rename − explain rename(2) errors

## SYNOPSIS

#include <libexplain/rename.h>
const char *explain_rename(const char *oldpath, const char *newpath);
const char *explain_errno_rename(int errnum, const char *oldpath, const char *newpath);
void explain_message_rename(char *message, int message_size, const char *oldpath, const char *newpath);
void explain_message_errno_rename(char *message, int message_size, int errnum, const char *oldpath, const char *newpath);

## DESCRIPTION

The functions declared in the `<libexplain/rename.h>` include file may be used to explain errors returned by the *rename*(2) system call.

### explain_rename

const char *explain_rename(const char *oldpath, const char *newpath);

The explain_rename function is used to obtain an explanation of an error returned by the *rename*(2) function. The least the message will contain is the value of strerror(errno), but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (rename(oldpath, rewpath) < 0)
{
    fprintf(stderr, "%s\n", explain_rename(oldpath, newpath));
    exit(EXIT_FAILURE);
}
```

*oldpath* The original oldpath, exactly as passed to the *rename*(2) system call.

*newpath* The original newpath, exactly as passed to the *rename*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_rename

const char *explain_errno_rename(int errnum, const char *oldpath, const char *newpath);

The explain_errno_rename function is used to obtain an explanation of an error returned by the rename(2) function. The least the message will contain is the value of strerror(errnum), but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (rename(oldpath, newpath) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_rename(err, oldpath,
        newpath));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*oldpath* The original oldpath, exactly as passed to the *rename*(2) system call.

*newpath* The original newpath, exactly as passed to the *rename*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_rename**

void explain_message_rename(char *message, int message_size, const char *oldpath, const char *newpath);

The explain_message_rename function is used to obtain an explanation of an error returned by the *rename*(2) function. The least the message will contain is the value of strerror(errno), but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (rename(oldpath, newpath) < 0)
{
    char message[3000];
    explain_message_rename(message, sizeof(message), oldpath,
        newpath);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe, if the buffer is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*oldpath* The original oldpath, exactly as passed to the *rename*(2) system call.

*newpath* The original newpath, exactly as passed to the *rename*(2) system call.

**explain_message_errno_rename**

void explain_message_errno_rename(char *message, int message_size, int errnum, const char *oldpath, const char *newpath);

The explain_message_errno_rename function is used to obtain an explanation of an error returned by the *rename*(2) function. The least the message will contain is the value of strerror(errnum), but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (rename(oldpath, newpath) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_rename(message, sizeof(message), err,
        oldpath, newpath);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe, given a thread safe buffer.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*oldpath* The original oldpath, exactly as passed to the *rename*(2) system call.

*newpath* The original newpath, exactly as passed to the *rename*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_rename_or_die − change the name of a file and report errors

## SYNOPSIS

#include <libexplain/rename.h>

void explain_rename_or_die(const char *oldpath, const char *newpath);

## DESCRIPTION

The **explain_rename_or_die** function is used to call the *rename*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_rename*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_rename_or_die(oldpath, newpath);
```

*oldpath* The oldpath, exactly as to be passed to the *rename*(2) system call.

*newpath* The newpath, exactly as to be passed to the *rename*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*rename*(2)
change the name or location of a file

*explain_rename*(3)
explain *rename*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_rmdir − explain rmdir(2) errors

## SYNOPSIS

#include <libexplain/rmdir.h>

const char *explain_rmdir(const char *pathname);
const char *explain_errno_rmdir(int errnum, const char pathname);
void explain_message_rmdir(char *message, int message_size, const char *pathname);
void explain_message_errno_rmdir(char *message, int message_size, int errnum, const char *pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *rmdir*(2) system call.

### explain_rmdir

const char *explain_rmdir(const char *pathname);

The **explain_rmdir** function may be used to describe errors returned by the *rmdir*() system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (rmdir(pathname) < 0)
{
    fprintf(stderr, "%s\n", explain_rmdir(pathname));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *rmdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_rmdir

const char *explain_errno_rmdir(int errnum, const char *pathname);

The **explain_errno_rmdir** function may be used to describe errors returned by the *rmdir*() system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (rmdir(pathname) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_rmdir(err, pathname));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *rmdir*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_rmdir**

void explain_message_rmdir(char *message, int message_size, const char *pathname);

The **explain_message_rmdir** function may be used to describe errors returned by the *rmdir*() system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (rmdir(pathname) < 0)
{
    char message[3000];
    explain_message_rmdir(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *rmdir*(2) system call.

**explain_message_errno_rmdir**

void explain_message_errno_rmdir(char *message, int message_size, int errnum, const char *pathname);

The **explain_message_errno_rmdir** function may be used to describe errors returned by the *rmdir*() system call. The least the message will contain is the value of *strerror(errnum)*, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (rmdir(pathname) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_rmdir(message, sizeof(message), err, pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *rmdir*(2) system call.

## SEE ALSO

*rmdir* delete a directory

*explain_rmdir_or_die*
delete a directory and report errors

## COPYRIGHT

## NAME

explain_rmdir_or_die − delete a directory and report errors

## SYNOPSIS

#include <libexplain/rmdir.h>

void explain_rmdir_or_die(const char *pathname);

## DESCRIPTION

The **explain_rmdir_or_die** function is used to call the *rmdir*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_rmdir*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_rmdir_or_die(pathname);
```

*pathname*
: The pathname, exactly as to be passed to the *rmdir*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*rmdir*(2) delete a directory

*explain_rmdir*(3)
: explain *rmdir*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_select − explain select(2) errors

## SYNOPSIS

#include <sys/select.h> #include <libexplain/select.h>

const char *explain_select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
const char *explain_errno_select(int errnum, int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
void explain_message_select(char *message, int message_size, int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
void explain_message_errno_select(char *message, int message_size, int errnum, int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *select*(2) system call.

### explain_select

const char *explain_select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

The **explain_select** function is used to obtain an explanation of an error returned by the *select*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (select(nfds, readfds, writefds, exceptfds, timeout) < 0)
{
    fprintf(stderr, "%s\n", explain_select(nfds,
        readfds, writefds, exceptfds, timeout));
    exit(EXIT_FAILURE);
}
```

*nfds* The original nfds, exactly as passed to the *select*(2) system call.

*readfds* The original readfds, exactly as passed to the *select*(2) system call.

*writefds* The original writefds, exactly as passed to the *select*(2) system call.

*exceptfds*
The original exceptfds, exactly as passed to the *select*(2) system call.

*timeout* The original timeout, exactly as passed to the *select*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_select

const char *explain_errno_select(int errnum, int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

The **explain_errno_select** function is used to obtain an explanation of an error returned by the *select*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (select(nfds, readfds, writefds, exceptfds, timeout) < 0)
```

```
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_select(err,
        nfds, readfds, writefds, exceptfds, timeout));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nfds* The original nfds, exactly as passed to the *select*(2) system call.

*readfds* The original readfds, exactly as passed to the *select*(2) system call.

*writefds* The original writefds, exactly as passed to the *select*(2) system call.

*exceptfds*
The original exceptfds, exactly as passed to the *select*(2) system call.

*timeout* The original timeout, exactly as passed to the *select*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_select**

void explain_message_select(char *message, int message_size, int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

The **explain_message_select** function may be used to obtain an explanation of an error returned by the *select*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (select(nfds, readfds, writefds, exceptfds, timeout) < 0)
{
    char message[3000];
    explain_message_select(message, sizeof(message),
        nfds, readfds, writefds, exceptfds, timeout);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nfds* The original nfds, exactly as passed to the *select*(2) system call.

*readfds* The original readfds, exactly as passed to the *select*(2) system call.

*writefds* The original writefds, exactly as passed to the *select*(2) system call.

*exceptfds*
The original exceptfds, exactly as passed to the *select*(2) system call.

*timeout* The original timeout, exactly as passed to the *select*(2) system call.

**explain_message_errno_select**

void explain_message_errno_select(char *message, int message_size, int errnum, int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

The **explain_message_errno_select** function may be used to obtain an explanation of an error returned by the *select*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (select(nfds, readfds, writefds, exceptfds, timeout) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_select(message, sizeof(message), err,
        nfds, readfds, writefds, exceptfds, timeout);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nfds* The original nfds, exactly as passed to the *select*(2) system call.

*readfds* The original readfds, exactly as passed to the *select*(2) system call.

*writefds* The original writefds, exactly as passed to the *select*(2) system call.

*exceptfds*
The original exceptfds, exactly as passed to the *select*(2) system call.

*timeout* The original timeout, exactly as passed to the *select*(2) system call.

## SEE ALSO

*select*(2) blah blah

*explain_select_or_die*(3)
blah blah and report errors

## COPYRIGHT

## NAME

explain_select_or_die − blah blah and report errors

## SYNOPSIS

#include <libexplain/select.h>

void explain_select_or_die(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

## DESCRIPTION

The **explain_select_or_die** function is used to call the *select*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_select*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_select_or_die(nfds, readfds, writefds, exceptfds, timeout);
```

*nfds* The nfds, exactly as to be passed to the *select*(2) system call.

*readfds* The readfds, exactly as to be passed to the *select*(2) system call.

*writefds* The writefds, exactly as to be passed to the *select*(2) system call.

*exceptfds*
The exceptfds, exactly as to be passed to the *select*(2) system call.

*timeout* The timeout, exactly as to be passed to the *select*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*select*(2) blah blah

*explain_select*(3)
explain *select*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setbuf − explain *setbuf*(3) errors

## SYNOPSIS

#include <libexplain/setbuf.h>

const char *explain_setbuf(FILE *fp, char *data);
const char *explain_errno_setbuf(int errnum, FILE *fp, char *data);
void explain_message_setbuf(char *message, int message_size, FILE *fp, char *data);
void explain_message_errno_setbuf(char *message, int message_size, int errnum, FILE *fp, char *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setbuf*(3) system call.

### explain_setbuf

const char *explain_setbuf(FILE *fp, char *data);

The **explain_setbuf** function is used to obtain an explanation of an error returned by the *setbuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *setbuf*(3) system call.

*data* The original data, exactly as passed to the *setbuf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void result = setbuf(fp, data);
if (result < 0 && errno != 0)
{
   fprintf(stderr, "%s\n", explain_setbuf(fp, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setbuf_or_die*(3) function.

### explain_errno_setbuf

const char *explain_errno_setbuf(int errnum, FILE *fp, char *data);

The **explain_errno_setbuf** function is used to obtain an explanation of an error returned by the *setbuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *setbuf*(3) system call.

*data* The original data, exactly as passed to the *setbuf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void result = setbuf(fp, data);
if (result < 0 && errno != 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setbuf(err, fp, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setbuf_or_die*(3) function.

**explain_message_setbuf**

void explain_message_setbuf(char *message, int message_size, FILE *fp, char *data);

The **explain_message_setbuf** function is used to obtain an explanation of an error returned by the *setbuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *setbuf*(3) system call.

*data* The original data, exactly as passed to the *setbuf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void result = setbuf(fp, data);
if (result < 0 && errno != 0)
{
    char message[3000];
   explain_message_setbuf(message, sizeof(message), fp, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setbuf_or_die*(3) function.

**explain_message_errno_setbuf**

void explain_message_errno_setbuf(char *message, int message_size, int errnum, FILE *fp, char *data);

The **explain_message_errno_setbuf** function is used to obtain an explanation of an error returned by the *setbuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *setbuf*(3) system call.

*data* The original data, exactly as passed to the *setbuf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void result = setbuf(fp, data);
if (result < 0 && errno != 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setbuf(message, sizeof(message), err,
   fp, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setbuf_or_die*(3) function.

## SEE ALSO

*setbuf*(3)
set stream buffer

*explain_setbuf_or_die*(3)
set stream buffer and report errors

## COPYRIGHT

## NAME

explain_setbuffer − explain *setbuffer*(3) errors

## SYNOPSIS

#include <libexplain/setbuffer.h>

const char *explain_setbuffer(FILE *fp, char *data, size_t size);
const char *explain_errno_setbuffer(int errnum, FILE *fp, char *data, size_t size);
void explain_message_setbuffer(char *message, int message_size, FILE *fp, char *data, size_t size);
void explain_message_errno_setbuffer(char *message, int message_size, int errnum, FILE *fp, char *data, size_t size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setbuffer*(3) system call.

### explain_setbuffer

const char *explain_setbuffer(FILE *fp, char *data, size_t size);

The **explain_setbuffer** function is used to obtain an explanation of an error returned by the *setbuffer*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *setbuffer*(3) system call.

*data* The original data, exactly as passed to the *setbuffer*(3) system call.

*size* The original size, exactly as passed to the *setbuffer*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void result = setbuffer(fp, data, size);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_setbuffer(fp, data, size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setbuffer_or_die*(3) function.

### explain_errno_setbuffer

const char *explain_errno_setbuffer(int errnum, FILE *fp, char *data, size_t size);

The **explain_errno_setbuffer** function is used to obtain an explanation of an error returned by the *setbuffer*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *setbuffer*(3) system call.

*data* The original data, exactly as passed to the *setbuffer*(3) system call.

*size* The original size, exactly as passed to the *setbuffer*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void result = setbuffer(fp, data, size);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setbuffer(err, fp, data,
   size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setbuffer_or_die*(3) function.

**explain_message_setbuffer**

void explain_message_setbuffer(char *message, int message_size, FILE *fp, char *data, size_t size);

The **explain_message_setbuffer** function is used to obtain an explanation of an error returned by the *setbuffer*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *setbuffer*(3) system call.

*data* The original data, exactly as passed to the *setbuffer*(3) system call.

*size* The original size, exactly as passed to the *setbuffer*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void result = setbuffer(fp, data, size);
if (result < 0)
{
    char message[3000];
   explain_message_setbuffer(message, sizeof(message), fp, data,
   size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setbuffer_or_die*(3) function.

**explain_message_errno_setbuffer**

void explain_message_errno_setbuffer(char *message, int message_size, int errnum, FILE *fp, char *data, size_t size);

The **explain_message_errno_setbuffer** function is used to obtain an explanation of an error returned by the *setbuffer*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *setbuffer*(3) system call.

*data* The original data, exactly as passed to the *setbuffer*(3) system call.

*size* The original size, exactly as passed to the *setbuffer*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void result = setbuffer(fp, data, size);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setbuffer(message, sizeof(message), err,
   fp, data, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setbuffer_or_die*(3) function.

## SEE ALSO

*setbuffer*(3)
stream buffering operations

*explain_setbuffer_or_die*(3)
stream buffering operations and report errors

## COPYRIGHT

## NAME

explain_setbuffer_or_die − stream buffering operations and report errors

## SYNOPSIS

#include <libexplain/setbuffer.h>

void explain_setbuffer_or_die(FILE *fp, char *data, size_t size);
void explain_setbuffer_on_error(FILE *fp, char *data, size_t size);

## DESCRIPTION

The **explain_setbuffer_or_die** function is used to call the *setbuffer*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setbuffer*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setbuffer_on_error** function is used to call the *setbuffer*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setbuffer*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *setbuffer*(3) system call.

*data* The data, exactly as to be passed to the *setbuffer*(3) system call.

*size* The size, exactly as to be passed to the *setbuffer*(3) system call.

## RETURN VALUE

The **explain_setbuffer_or_die** function only returns on success, see *setbuffer*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setbuffer_on_error** function always returns the value return by the wrapped *setbuffer*(3) system call.

## EXAMPLE

The **explain_setbuffer_or_die** function is intended to be used in a fashion similar to the following example:

```
void result = explain_setbuffer_or_die(fp, data, size);
```

## SEE ALSO

*setbuffer*(3)
stream buffering operations

*explain_setbuffer*(3)
explain *setbuffer*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setbuf_or_die − set stream buffer and report errors

## SYNOPSIS

#include <libexplain/setbuf.h>

void explain_setbuf_or_die(FILE *fp, char *data);
void explain_setbuf_on_error(FILE *fp, char *data);

## DESCRIPTION

The **explain_setbuf_or_die** function is used to call the *setbuf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setbuf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setbuf_on_error** function is used to call the *setbuf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setbuf*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *setbuf*(3) system call.

*data* The data, exactly as to be passed to the *setbuf*(3) system call.

## RETURN VALUE

The **explain_setbuf_or_die** function only returns on success, see *setbuf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setbuf_on_error** function always returns the value return by the wrapped *setbuf*(3) system call.

## EXAMPLE

The **explain_setbuf_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setbuf_or_die(fp, data);
```

## SEE ALSO

*setbuf*(3)
set stream buffer

*explain_setbuf*(3)
explain *setbuf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setdomainname − explain setdomainname(2) errors

## SYNOPSIS

#include <libexplain/setdomainname.h>

const char *explain_setdomainname(const char *data, size_t data_size);
const char *explain_errno_setdomainname(int errnum, const char *data, size_t data_size);
void explain_message_setdomainname(char *message, int message_size, const char *data, size_t data_size);
void explain_message_errno_setdomainname(char *message, int message_size, int errnum, const char *data, size_t data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setdomainname*(2) system call.

### explain_setdomainname

const char *explain_setdomainname(const char *data, size_t data_size);

The **explain_setdomainname** function is used to obtain an explanation of an error returned by the *setdomainname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *setdomainname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *setdomainname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setdomainname(data, data_size) < 0)
{
    fprintf(stderr, "%s\n", explain_setdomainname(data,
    data_size));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setdomainname_or_die*(3) function.

### explain_errno_setdomainname

const char *explain_errno_setdomainname(int errnum, const char *data, size_t data_size);

The **explain_errno_setdomainname** function is used to obtain an explanation of an error returned by the *setdomainname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *setdomainname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *setdomainname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setdomainname(data, data_size) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setdomainname(err, data,
   data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setdomainname_or_die*(3) function.

**explain_message_setdomainname**

void explain_message_setdomainname(char *message, int message_size, const char *data, size_t data_size);

The **explain_message_setdomainname** function is used to obtain an explanation of an error returned by the *setdomainname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *setdomainname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *setdomainname*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setdomainname(data, data_size) < 0)
{
    char message[3000];
   explain_message_setdomainname(message, sizeof(message), data,
   data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setdomainname_or_die*(3) function.

**explain_message_errno_setdomainname**

void explain_message_errno_setdomainname(char *message, int message_size, int errnum, const char *data, size_t data_size);

The **explain_message_errno_setdomainname** function is used to obtain an explanation of an error returned by the *setdomainname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *setdomainname*(2) system call.

*data_size*
The original data_size, exactly as passed to the *setdomainname*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setdomainname(data, data_size) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setdomainname(message, sizeof(message),
   err, data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setdomainname_or_die*(3) function.

## SEE ALSO

*setdomainname*(2)
set domain name

*explain_setdomainname_or_die*(3)
set domain name and report errors

## COPYRIGHT

## NAME

explain_setdomainname_or_die − set domain name and report errors

## SYNOPSIS

#include <libexplain/setdomainname.h>

void explain_setdomainname_or_die(const char *data, size_t data_size);
int explain_setdomainname_on_error(const char *data, size_t data_size);

## DESCRIPTION

The **explain_setdomainname_or_die** function is used to call the *setdomainname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setdomainname*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setdomainname_on_error** function is used to call the *setdomainname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setdomainname*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *setdomainname*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *setdomainname*(2) system call.

## RETURN VALUE

The **explain_setdomainname_or_die** function only returns on success, see *setdomainname*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setdomainname_on_error** function always returns the value return by the wrapped *setdomainname*(2) system call.

## EXAMPLE

The **explain_setdomainname_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setdomainname_or_die(data, data_size);
```

## SEE ALSO

*setdomainname*(2)
set domain name

*explain_setdomainname*(3)
explain *setdomainname*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setenv − explain *setenv*(3) errors

## SYNOPSIS

#include <libexplain/setenv.h>

const char *explain_setenv(const char *name, const char *value, int overwrite);
const char *explain_errno_setenv(int errnum, const char *name, const char *value, int overwrite);
void explain_message_setenv(char *message, int message_size, const char *name, const char *value, int overwrite);
void explain_message_errno_setenv(char *message, int message_size, int errnum, const char *name, const char *value, int overwrite);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setenv*(3) system call.

### explain_setenv

const char *explain_setenv(const char *name, const char *value, int overwrite);

The **explain_setenv** function is used to obtain an explanation of an error returned by the *setenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*name* The original name, exactly as passed to the *setenv*(3) system call.

*value* The original value, exactly as passed to the *setenv*(3) system call.

*overwrite*
The original overwrite, exactly as passed to the *setenv*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setenv(name, value, overwrite) < 0)
{
    fprintf(stderr, "%s\n", explain_setenv(name, value,
    overwrite));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setenv_or_die*(3) function.

### explain_errno_setenv

const char *explain_errno_setenv(int errnum, const char *name, const char *value, int overwrite);

The **explain_errno_setenv** function is used to obtain an explanation of an error returned by the *setenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*name* The original name, exactly as passed to the *setenv*(3) system call.

*value* The original value, exactly as passed to the *setenv*(3) system call.

*overwrite*
The original overwrite, exactly as passed to the *setenv*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setenv(name, value, overwrite) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setenv(err, name, value,
   overwrite));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setenv_or_die*(3) function.

**explain_message_setenv**

void explain_message_setenv(char *message, int message_size, const char *name, const char *value, int overwrite);

The **explain_message_setenv** function is used to obtain an explanation of an error returned by the *setenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*name* The original name, exactly as passed to the *setenv*(3) system call.

*value* The original value, exactly as passed to the *setenv*(3) system call.

*overwrite*
The original overwrite, exactly as passed to the *setenv*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setenv(name, value, overwrite) < 0)
{
    char message[3000];
   explain_message_setenv(message, sizeof(message), name, value,
   overwrite);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setenv_or_die*(3) function.

**explain_message_errno_setenv**

void explain_message_errno_setenv(char *message, int message_size, int errnum, const char *name, const char *value, int overwrite);

The **explain_message_errno_setenv** function is used to obtain an explanation of an error returned by the *setenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*name* The original name, exactly as passed to the *setenv*(3) system call.

*value* The original value, exactly as passed to the *setenv*(3) system call.

*overwrite*
The original overwrite, exactly as passed to the *setenv*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setenv(name, value, overwrite) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setenv(message, sizeof(message), err,
   name, value, overwrite);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setenv_or_die*(3) function.

## SEE ALSO

*setenv*(3)
change or add an environment variable

*explain_setenv_or_die*(3)
change or add an environment variable and report errors

## COPYRIGHT

## NAME

explain_setenv_or_die − change or add an environment variable and report errors

## SYNOPSIS

#include <libexplain/setenv.h>

void explain_setenv_or_die(const char *name, const char *value, int overwrite);
int explain_setenv_on_error(const char *name, const char *value, int overwrite);

## DESCRIPTION

The **explain_setenv_or_die** function is used to call the *setenv*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setenv*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setenv_on_error** function is used to call the *setenv*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setenv*(3) function, but still returns to the caller.

*name* The name, exactly as to be passed to the *setenv*(3) system call.

*value* The value, exactly as to be passed to the *setenv*(3) system call.

*overwrite*
The overwrite, exactly as to be passed to the *setenv*(3) system call.

## RETURN VALUE

The **explain_setenv_or_die** function only returns on success, see *setenv*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setenv_on_error** function always returns the value return by the wrapped *setenv*(3) system call.

## EXAMPLE

The **explain_setenv_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setenv_or_die(name, value, overwrite);
```

## SEE ALSO

*setenv*(3)
change or add an environment variable

*explain_setenv*(3)
explain *setenv*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setgid − explain *setgid*(2) errors

## SYNOPSIS

#include <libexplain/setgid.h>

const char *explain_setgid(gid_t gid);
const char *explain_errno_setgid(int errnum, gid_t gid);
void explain_message_setgid(char *message, int message_size, gid_t gid);
void explain_message_errno_setgid(char *message, int message_size, int errnum, gid_t gid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setgid*(2) system call.

### explain_setgid

const char *explain_setgid(gid_t gid);

The **explain_setgid** function is used to obtain an explanation of an error returned by the *setgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*gid* The original gid, exactly as passed to the *setgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setgid(gid) < 0)
{
   fprintf(stderr, "%s\n", explain_setgid(gid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setgid_or_die*(3) function.

### explain_errno_setgid

const char *explain_errno_setgid(int errnum, gid_t gid);

The **explain_errno_setgid** function is used to obtain an explanation of an error returned by the *setgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*gid* The original gid, exactly as passed to the *setgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setgid(gid) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_setgid(err, gid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setgid_or_die*(3) function.

**explain_message_setgid**

void explain_message_setgid(char *message, int message_size, gid_t gid);

The **explain_message_setgid** function is used to obtain an explanation of an error returned by the *setgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*gid* The original gid, exactly as passed to the *setgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setgid(gid) < 0)
{
    char message[3000];
    explain_message_setgid(message, sizeof(message), gid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setgid_or_die*(3) function.

**explain_message_errno_setgid**

void explain_message_errno_setgid(char *message, int message_size, int errnum, gid_t gid);

The **explain_message_errno_setgid** function is used to obtain an explanation of an error returned by the *setgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*gid* The original gid, exactly as passed to the *setgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setgid(gid) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_setgid(message, sizeof(message), err,
    gid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setgid_or_die*(3) function.

## SEE ALSO

*setgid*(2)
set group identity

*explain_setgid_or_die*(3)
set group identity and report errors

## COPYRIGHT

## NAME

explain_setgid_or_die − set group identity and report errors

## SYNOPSIS

#include <libexplain/setgid.h>

void explain_setgid_or_die(gid_t gid);
int explain_setgid_on_error(gid_t gid);

## DESCRIPTION

The **explain_setgid_or_die** function is used to call the *setgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setgid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setgid_on_error** function is used to call the *setgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setgid*(3) function, but still returns to the caller.

*gid* The gid, exactly as to be passed to the *setgid*(2) system call.

## RETURN VALUE

The **explain_setgid_or_die** function only returns on success, see *setgid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setgid_on_error** function always returns the value return by the wrapped *setgid*(2) system call.

## EXAMPLE

The **explain_setgid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setgid_or_die(gid);
```

## SEE ALSO

*setgid*(2)
set group identity

*explain_setgid*(3)
explain *setgid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setgroups − explain setgroups(2) errors

## SYNOPSIS

#include <libexplain/setgroups.h>

const char *explain_setgroups(size_t data_size, const gid_t *data);
const char *explain_errno_setgroups(int errnum, size_t data_size, const gid_t *data);
void explain_message_setgroups(char *message, int message_size, size_t data_size, const gid_t *data);
void explain_message_errno_setgroups(char *message, int message_size, int errnum, size_t data_size, const gid_t *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setgroups*(2) system call.

### explain_setgroups

const char *explain_setgroups(size_t data_size, const gid_t *data);

The **explain_setgroups** function is used to obtain an explanation of an error returned by the *setgroups*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data_size*
: The original data_size, exactly as passed to the *setgroups*(2) system call.

*data*
: The original data, exactly as passed to the *setgroups*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setgroups(data_size, data) < 0)
{
   fprintf(stderr, "%s\n", explain_setgroups(data_size, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setgroups_or_die*(3) function.

### explain_errno_setgroups

const char *explain_errno_setgroups(int errnum, size_t data_size, const gid_t *data);

The **explain_errno_setgroups** function is used to obtain an explanation of an error returned by the *setgroups*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data_size*
: The original data_size, exactly as passed to the *setgroups*(2) system call.

*data*
: The original data, exactly as passed to the *setgroups*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setgroups(data_size, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setgroups(err,
   data_size, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setgroups_or_die*(3) function.

**explain_message_setgroups**

void explain_message_setgroups(char *message, int message_size, size_t data_size, const gid_t *data);

The **explain_message_setgroups** function is used to obtain an explanation of an error returned by the *setgroups*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data_size*
The original data_size, exactly as passed to the *setgroups*(2) system call.

*data* The original data, exactly as passed to the *setgroups*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setgroups(data_size, data) < 0)
{
    char message[3000];
   explain_message_setgroups(message, sizeof(message), data_size,
   data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setgroups_or_die*(3) function.

**explain_message_errno_setgroups**

void explain_message_errno_setgroups(char *message, int message_size, int errnum, size_t data_size, const gid_t *data);

The **explain_message_errno_setgroups** function is used to obtain an explanation of an error returned by the *setgroups*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data_size*
The original data_size, exactly as passed to the *setgroups*(2) system call.

*data* The original data, exactly as passed to the *setgroups*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setgroups(data_size, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setgroups(message, sizeof(message), err,
   data_size, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setgroups_or_die*(3) function.

## SEE ALSO

*setgroups*(2)
get/set list of supplementary group IDs

*explain_setgroups_or_die*(3)
get/set list of supplementary group IDs and report errors

## COPYRIGHT

## NAME

explain_setgroups_or_die − set supplementary group IDs and report errors

## SYNOPSIS

#include <libexplain/setgroups.h>

void explain_setgroups_or_die(size_t data_size, const gid_t *data);
int explain_setgroups_on_error(size_t data_size, const gid_t *data);

## DESCRIPTION

The **explain_setgroups_or_die** function is used to call the *setgroups*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setgroups*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setgroups_on_error** function is used to call the *setgroups*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setgroups*(3) function, but still returns to the caller.

*data_size*
: The data_size, exactly as to be passed to the *setgroups*(2) system call.

*data*
: The data, exactly as to be passed to the *setgroups*(2) system call.

## RETURN VALUE

The **explain_setgroups_or_die** function only returns on success, see *setgroups*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setgroups_on_error** function always returns the value return by the wrapped *setgroups*(2) system call.

## EXAMPLE

The **explain_setgroups_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setgroups_or_die(data_size, data);
```

## SEE ALSO

*setgroups*(2)
: get/set list of supplementary group IDs

*explain_setgroups*(3)
: explain *setgroups*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_sethostname − explain sethostname(2) errors

## SYNOPSIS

#include <libexplain/sethostname.h>

const char *explain_sethostname(const char *name, size_t name_size);
const char *explain_errno_sethostname(int errnum, const char *name, size_t name_size);
void explain_message_sethostname(char *message, int message_size, const char *name, size_t name_size);
void explain_message_errno_sethostname(char *message, int message_size, int errnum, const char *name, size_t name_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *sethostname*(2) system call.

### explain_sethostname

const char *explain_sethostname(const char *name, size_t name_size);

The **explain_sethostname** function is used to obtain an explanation of an error returned by the *sethostname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (sethostname(name, name_size) < 0)
{
    fprintf(stderr, "%s\n", explain_sethostname(name, name_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_sethostname_or_die*(3) function.

*name* The original name, exactly as passed to the *sethostname*(2) system call.

*name_size*
The original name_size, exactly as passed to the *sethostname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_sethostname

const char *explain_errno_sethostname(int errnum, const char *name, size_t name_size);

The **explain_errno_sethostname** function is used to obtain an explanation of an error returned by the *sethostname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (sethostname(name, name_size) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_sethostname(err, name, name_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_sethostname_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*name* The original name, exactly as passed to the *sethostname*(2) system call.

*name_size*
The original name_size, exactly as passed to the *sethostname*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_sethostname**

void explain_message_sethostname(char *message, int message_size, const char *name, size_t name_size);

The **explain_message_sethostname** function is used to obtain an explanation of an error returned by the *sethostname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (sethostname(name, name_size) < 0)
{
    char message[3000];
    explain_message_sethostname(message, sizeof(message), name, name_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_sethostname_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*name* The original name, exactly as passed to the *sethostname*(2) system call.

*name_size*
The original name_size, exactly as passed to the *sethostname*(2) system call.

**explain_message_errno_sethostname**

void explain_message_errno_sethostname(char *message, int message_size, int errnum, const char *name, size_t name_size);

The **explain_message_errno_sethostname** function is used to obtain an explanation of an error returned by the *sethostname*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (sethostname(name, name_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_sethostname(message, sizeof(message), err, name,
        name_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_sethostname_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*name* The original name, exactly as passed to the *sethostname*(2) system call.

*name_size*
The original name_size, exactly as passed to the *sethostname*(2) system call.

## SEE ALSO

*sethostname*(2)
get/set hostname

*explain_sethostname_or_die*(3)
get/set hostname and report errors

## COPYRIGHT

## NAME

explain_sethostname_or_die − get/set hostname and report errors

## SYNOPSIS

#include <libexplain/sethostname.h>

void explain_sethostname_or_die(const char *name, size_t name_size);
intexplain_sethostname_on_error(const char *name, size_t name_size);

## DESCRIPTION

The **explain_sethostname_or_die** function is used to call the *sethostname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_sethostname*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_sethostname_on_error** function is used to call the *sethostname*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_sethostname*(3) function, but still returns to the caller.

*name* The name, exactly as to be passed to the *sethostname*(2) system call.

*name_size*
The name_size, exactly as to be passed to the *sethostname*(2) system call.

## RETURN VALUE

The **explain_sethostname_or_die** function only returns on success, see *sethostname*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_sethostname_on_error** function always returns the value return by the wrapped *sethostname*(2) system call.

## EXAMPLE

The **explain_sethostname_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_sethostname_or_die(name, name_size);
```

## SEE ALSO

*sethostname*(2)
get/set hostname

*explain_sethostname*(3)
explain *sethostname*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setlinebuf − explain *setlinebuf*(3) errors

## SYNOPSIS

#include <libexplain/setlinebuf.h>

const char *explain_setlinebuf(FILE *fp);
const char *explain_errno_setlinebuf(int errnum, FILE *fp);
void explain_message_setlinebuf(char *message, int message_size, FILE *fp);
void explain_message_errno_setlinebuf(char *message, int message_size, int errnum, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setlinebuf*(3) system call.

### explain_setlinebuf

const char *explain_setlinebuf(FILE *fp);

The **explain_setlinebuf** function is used to obtain an explanation of an error returned by the *setlinebuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *setlinebuf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void result = setlinebuf(fp);
if (result < 0 && errno != 0)
{
   fprintf(stderr, "%s\n", explain_setlinebuf(fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setlinebuf_or_die*(3) function.

### explain_errno_setlinebuf

const char *explain_errno_setlinebuf(int errnum, FILE *fp);

The **explain_errno_setlinebuf** function is used to obtain an explanation of an error returned by the *setlinebuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *setlinebuf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void result = setlinebuf(fp);
if (result < 0 && errno != 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setlinebuf(err, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setlinebuf_or_die*(3) function.

**explain_message_setlinebuf**

void explain_message_setlinebuf(char *message, int message_size, FILE *fp);

The **explain_message_setlinebuf** function is used to obtain an explanation of an error returned by the *setlinebuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *setlinebuf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void result = setlinebuf(fp);
if (result < 0 && errno != 0)
{
    char message[3000];
   explain_message_setlinebuf(message, sizeof(message), fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setlinebuf_or_die*(3) function.

**explain_message_errno_setlinebuf**

void explain_message_errno_setlinebuf(char *message, int message_size, int errnum, FILE *fp);

The **explain_message_errno_setlinebuf** function is used to obtain an explanation of an error returned by the *setlinebuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *setlinebuf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
void result = setlinebuf(fp);
if (result < 0 && errno != 0)
```

```
{
    int err = errno;
    char message[3000];
   explain_message_errno_setlinebuf(message, sizeof(message),
   err, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setlinebuf_or_die*(3) function.

## SEE ALSO

*setlinebuf*(3)
stream buffering operations

*explain_setlinebuf_or_die*(3)
stream buffering operations and report errors

## COPYRIGHT

## NAME

explain_setlinebuf_or_die − stream buffering operations and report errors

## SYNOPSIS

#include <libexplain/setlinebuf.h>

void explain_setlinebuf_or_die(FILE *fp);
void explain_setlinebuf_on_error(FILE *fp);

## DESCRIPTION

The **explain_setlinebuf_or_die** function is used to call the *setlinebuf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setlinebuf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setlinebuf_on_error** function is used to call the *setlinebuf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setlinebuf*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *setlinebuf*(3) system call.

## RETURN VALUE

The **explain_setlinebuf_or_die** function only returns on success, see *setlinebuf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setlinebuf_on_error** function always returns the value return by the wrapped *setlinebuf*(3) system call.

## EXAMPLE

The **explain_setlinebuf_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setlinebuf_or_die(fp);
```

## SEE ALSO

*setlinebuf*(3)
stream buffering operations

*explain_setlinebuf*(3)
explain *setlinebuf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setpgid − explain *setpgid*(2) errors

## SYNOPSIS

#include <libexplain/setpgid.h>

const char *explain_setpgid(pid_t pid, pid_t pgid);
const char *explain_errno_setpgid(int errnum, pid_t pid, pid_t pgid);
void explain_message_setpgid(char *message, int message_size, pid_t pid, pid_t pgid);
void explain_message_errno_setpgid(char *message, int message_size, int errnum, pid_t pid, pid_t pgid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setpgid*(2) system call.

### explain_setpgid

const char *explain_setpgid(pid_t pid, pid_t pgid);

The **explain_setpgid** function is used to obtain an explanation of an error returned by the *setpgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pid* The original pid, exactly as passed to the *setpgid*(2) system call.

*pgid* The original pgid, exactly as passed to the *setpgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setpgid(pid, pgid) < 0)
{
   fprintf(stderr, "%s\n", explain_setpgid(pid, pgid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setpgid_or_die*(3) function.

### explain_errno_setpgid

const char *explain_errno_setpgid(int errnum, pid_t pid, pid_t pgid);

The **explain_errno_setpgid** function is used to obtain an explanation of an error returned by the *setpgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *setpgid*(2) system call.

*pgid* The original pgid, exactly as passed to the *setpgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setpgid(pid, pgid) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setpgid(err, pid,
   pgid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setpgid_or_die*(3) function.

**explain_message_setpgid**

void explain_message_setpgid(char *message, int message_size, pid_t pid, pid_t pgid);

The **explain_message_setpgid** function is used to obtain an explanation of an error returned by the *setpgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pid* The original pid, exactly as passed to the *setpgid*(2) system call.

*pgid* The original pgid, exactly as passed to the *setpgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setpgid(pid, pgid) < 0)
{
    char message[3000];
   explain_message_setpgid(message, sizeof(message), pid, pgid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setpgid_or_die*(3) function.

**explain_message_errno_setpgid**

void explain_message_errno_setpgid(char *message, int message_size, int errnum, pid_t pid, pid_t pgid);

The **explain_message_errno_setpgid** function is used to obtain an explanation of an error returned by the *setpgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *setpgid*(2) system call.

*pgid* The original pgid, exactly as passed to the *setpgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setpgid(pid, pgid) < 0)
{
    int err = errno;
```

```
    char message[3000];
    explain_message_errno_setpgid(message, sizeof(message), err,
    pid, pgid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setpgid_or_die*(3) function.

## SEE ALSO

*setpgid*(2)
: set process group

*explain_setpgid_or_die*(3)
: set process group and report errors

## COPYRIGHT

## NAME

explain_setpgid_or_die − set process group and report errors

## SYNOPSIS

#include <libexplain/setpgid.h>

void explain_setpgid_or_die(pid_t pid, pid_t pgid);
int explain_setpgid_on_error(pid_t pid, pid_t pgid);

## DESCRIPTION

The **explain_setpgid_or_die** function is used to call the *setpgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setpgid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setpgid_on_error** function is used to call the *setpgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setpgid*(3) function, but still returns to the caller.

*pid* The pid, exactly as to be passed to the *setpgid*(2) system call.

*pgid* The pgid, exactly as to be passed to the *setpgid*(2) system call.

## RETURN VALUE

The **explain_setpgid_or_die** function only returns on success, see *setpgid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setpgid_on_error** function always returns the value return by the wrapped *setpgid*(2) system call.

## EXAMPLE

The **explain_setpgid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setpgid_or_die(pid, pgid);
```

## SEE ALSO

*setpgid*(2)
set process group

*explain_setpgid*(3)
explain *setpgid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setpgrp − explain *setpgrp*(2) errors

## SYNOPSIS

#include <libexplain/setpgrp.h>

const char *explain_setpgrp(pid_t pid, pid_t pgid);
const char *explain_errno_setpgrp(int errnum, pid_t pid, pid_t pgid);
void explain_message_setpgrp(char *message, int message_size, pid_t pid, pid_t pgid);
void explain_message_errno_setpgrp(char *message, int message_size, int errnum, pid_t pid, pid_t pgid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setpgrp*(2) system call.

**Note:** the *setpgrp*(2) function has two implementations. The System V version has no arguments, while the BSD version has two arguments. For simplicity of implementation, the argument list seen here includes the *pid* and *pgid* arguments.

The System V `getpgid()` semantics can be obtained by calling `setpgrp(0, 0)` on systems with the BSD version, and this is the API for libexplain, even on systems that do not use the BSD API.

### explain_setpgrp

const char *explain_setpgrp(pid_t pid, pid_t pgid);

The **explain_setpgrp** function is used to obtain an explanation of an error returned by the *setpgrp*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pid* The original pid, exactly as passed to the *setpgrp*(2) system call.

*pgid* The original pgid, exactly as passed to the *setpgrp*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setpgrp(pid, pgid) < 0)
{
    fprintf(stderr, "%s\n", explain_setpgrp(pid, pgid));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setpgrp_or_die*(3) function.

### explain_errno_setpgrp

const char *explain_errno_setpgrp(int errnum, pid_t pid, pid_t pgid);

The **explain_errno_setpgrp** function is used to obtain an explanation of an error returned by the *setpgrp*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *setpgrp*(2) system call.

*pgid* The original pgid, exactly as passed to the *setpgrp*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setpgrp(pid, pgid) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setpgrp(err, pid,
   pgid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setpgrp_or_die*(3) function.

**explain_message_setpgrp**

void explain_message_setpgrp(char *message, int message_size, pid_t pid, pid_t pgid);

The **explain_message_setpgrp** function is used to obtain an explanation of an error returned by the *setpgrp*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pid* The original pid, exactly as passed to the *setpgrp*(2) system call.

*pgid* The original pgid, exactly as passed to the *setpgrp*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setpgrp(pid, pgid) < 0)
{
    char message[3000];
   explain_message_setpgrp(message, sizeof(message), pid, pgid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setpgrp_or_die*(3) function.

**explain_message_errno_setpgrp**

void explain_message_errno_setpgrp(char *message, int message_size, int errnum, pid_t pid, pid_t pgid);

The **explain_message_errno_setpgrp** function is used to obtain an explanation of an error returned by the *setpgrp*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *setpgrp*(2) system call.

*pgid* The original pgid, exactly as passed to the *setpgrp*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setpgrp(pid, pgid) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setpgrp(message, sizeof(message), err,
   pid, pgid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setpgrp_or_die*(3) function.

## SEE ALSO

*setpgrp*(2)
set process group

*explain_setpgrp_or_die*(3)
set process group and report errors

## COPYRIGHT

## NAME

explain_setpgrp_or_die − set process group and report errors

## SYNOPSIS

#include <libexplain/setpgrp.h>

void explain_setpgrp_or_die(pid_t pid, pid_t pgid);
int explain_setpgrp_on_error(pid_t pid, pid_t pgid);

## DESCRIPTION

The **explain_setpgrp_or_die** function is used to call the *setpgrp*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setpgrp*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setpgrp_on_error** function is used to call the *setpgrp*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setpgrp*(3) function, but still returns to the caller.

*pid* The pid, exactly as to be passed to the *setpgrp*(2) system call.

*pgid* The pgid, exactly as to be passed to the *setpgrp*(2) system call.

**Note:** the *setpgrp*(2) function has two implementations. The System V version has no arguments, while the BSD version has two arguments. For simplicity of implementation, the argument list seen here includes the *pid* and *pgid* arguments.

The System V `getpgid()` semantics can be obtained by calling `setpgrp(0, 0)` on systems with the BSD version, and this is the API for libexplain, even on systems that do not use the BSD API.

## RETURN VALUE

The **explain_setpgrp_or_die** function only returns on success, see *setpgrp*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setpgrp_on_error** function always returns the value return by the wrapped *setpgrp*(2) system call.

## EXAMPLE

The **explain_setpgrp_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setpgrp_or_die(pid, pgid);
```

## SEE ALSO

*setpgrp*(2)
set process group

*explain_setpgrp*(3)
explain *setpgrp*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setregid − explain *setregid*(2) errors

## SYNOPSIS

#include <libexplain/setregid.h>

const char *explain_setregid(gid_t rgid, gid_t egid);
const char *explain_errno_setregid(int errnum, gid_t rgid, gid_t egid);
void explain_message_setregid(char *message, int message_size, gid_t rgid, gid_t egid);
void explain_message_errno_setregid(char *message, int message_size, int errnum, gid_t rgid, gid_t egid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setregid*(2) system call.

### explain_setregid

const char *explain_setregid(gid_t rgid, gid_t egid);

The **explain_setregid** function is used to obtain an explanation of an error returned by the *setregid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*rgid* The original rgid, exactly as passed to the *setregid*(2) system call.

*egid* The original egid, exactly as passed to the *setregid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setregid(rgid, egid) < 0)
{
   fprintf(stderr, "%s\n", explain_setregid(rgid, egid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setregid_or_die*(3) function.

### explain_errno_setregid

const char *explain_errno_setregid(int errnum, gid_t rgid, gid_t egid);

The **explain_errno_setregid** function is used to obtain an explanation of an error returned by the *setregid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*rgid* The original rgid, exactly as passed to the *setregid*(2) system call.

*egid* The original egid, exactly as passed to the *setregid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setregid(rgid, egid) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_setregid(err, rgid,
    egid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setregid_or_die*(3) function.

**explain_message_setregid**

void explain_message_setregid(char *message, int message_size, gid_t rgid, gid_t egid);

The **explain_message_setregid** function is used to obtain an explanation of an error returned by the *setregid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*rgid* The original rgid, exactly as passed to the *setregid*(2) system call.

*egid* The original egid, exactly as passed to the *setregid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setregid(rgid, egid) < 0)
{
    char message[3000];
    explain_message_setregid(message, sizeof(message), rgid,
    egid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setregid_or_die*(3) function.

**explain_message_errno_setregid**

void explain_message_errno_setregid(char *message, int message_size, int errnum, gid_t rgid, gid_t egid);

The **explain_message_errno_setregid** function is used to obtain an explanation of an error returned by the *setregid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*rgid* The original rgid, exactly as passed to the *setregid*(2) system call.

*egid* The original egid, exactly as passed to the *setregid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setregid(rgid, egid) < 0)
{
```

```
    int err = errno;
    char message[3000];
    explain_message_errno_setregid(message, sizeof(message), err,
    rgid, egid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setregid_or_die*(3) function.

## SEE ALSO

*setregid*(2)
: set real and/or effective group ID

*explain_setregid_or_die*(3)
: set real and/or effective group ID and report errors

## COPYRIGHT

## NAME

explain_setregid_or_die − set real and/or effective group ID and report errors

## SYNOPSIS

#include <libexplain/setregid.h>

void explain_setregid_or_die(gid_t rgid, gid_t egid);
int explain_setregid_on_error(gid_t rgid, gid_t egid);

## DESCRIPTION

The **explain_setregid_or_die** function is used to call the *setregid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setregid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setregid_on_error** function is used to call the *setregid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setregid*(3) function, but still returns to the caller.

*rgid* The rgid, exactly as to be passed to the *setregid*(2) system call.

*egid* The egid, exactly as to be passed to the *setregid*(2) system call.

## RETURN VALUE

The **explain_setregid_or_die** function only returns on success, see *setregid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setregid_on_error** function always returns the value return by the wrapped *setregid*(2) system call.

## EXAMPLE

The **explain_setregid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setregid_or_die(rgid, egid);
```

## SEE ALSO

*setregid*(2)
set real and/or effective group ID

*explain_setregid*(3)
explain *setregid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setresgid − explain *setresgid*(2) errors

## SYNOPSIS

#include <libexplain/setresgid.h>

const char *explain_setresgid(gid_t rgid, gid_t egid, gid_t sgid);
const char *explain_errno_setresgid(int errnum, gid_t rgid, gid_t egid, gid_t sgid);
void explain_message_setresgid(char *message, int message_size, gid_t rgid, gid_t egid, gid_t sgid);
void explain_message_errno_setresgid(char *message, int message_size, int errnum, gid_t rgid, gid_t egid, gid_t sgid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setresgid*(2) system call.

### explain_setresgid

const char *explain_setresgid(gid_t rgid, gid_t egid, gid_t sgid);

The **explain_setresgid** function is used to obtain an explanation of an error returned by the *setresgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*rgid* The original rgid, exactly as passed to the *setresgid*(2) system call.

*egid* The original egid, exactly as passed to the *setresgid*(2) system call.

*sgid* The original sgid, exactly as passed to the *setresgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setresgid(rgid, egid, sgid) < 0)
{
   fprintf(stderr, "%s\n", explain_setresgid(rgid, egid, sgid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setresgid_or_die*(3) function.

### explain_errno_setresgid

const char *explain_errno_setresgid(int errnum, gid_t rgid, gid_t egid, gid_t sgid);

The **explain_errno_setresgid** function is used to obtain an explanation of an error returned by the *setresgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*rgid* The original rgid, exactly as passed to the *setresgid*(2) system call.

*egid* The original egid, exactly as passed to the *setresgid*(2) system call.

*sgid* The original sgid, exactly as passed to the *setresgid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setresgid(rgid, egid, sgid) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setresgid(err, rgid,
   egid, sgid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setresgid_or_die*(3) function.

**explain_message_setresgid**

void explain_message_setresgid(char *message, int message_size, gid_t rgid, gid_t egid, gid_t sgid);

The **explain_message_setresgid** function is used to obtain an explanation of an error returned by the *setresgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*rgid* The original rgid, exactly as passed to the *setresgid*(2) system call.

*egid* The original egid, exactly as passed to the *setresgid*(2) system call.

*sgid* The original sgid, exactly as passed to the *setresgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setresgid(rgid, egid, sgid) < 0)
{
    char message[3000];
   explain_message_setresgid(message, sizeof(message), rgid,
   egid, sgid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setresgid_or_die*(3) function.

**explain_message_errno_setresgid**

void explain_message_errno_setresgid(char *message, int message_size, int errnum, gid_t rgid, gid_t egid, gid_t sgid);

The **explain_message_errno_setresgid** function is used to obtain an explanation of an error returned by the *setresgid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*rgid* The original rgid, exactly as passed to the *setresgid*(2) system call.

*egid* The original egid, exactly as passed to the *setresgid*(2) system call.

*sgid* The original sgid, exactly as passed to the *setresgid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setresgid(rgid, egid, sgid) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setresgid(message, sizeof(message), err,
   rgid, egid, sgid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setresgid_or_die*(3) function.

## SEE ALSO

*setresgid*(2)
set real, effective and saved group ID

*explain_setresgid_or_die*(3)
set real, effective and saved group ID and report errors

## COPYRIGHT

## NAME

explain_setresgid_or_die − set r/e/s group ID and report errors

## SYNOPSIS

#include <libexplain/setresgid.h>

void explain_setresgid_or_die(gid_t rgid, gid_t egid, gid_t sgid);
int explain_setresgid_on_error(gid_t rgid, gid_t egid, gid_t sgid);

## DESCRIPTION

The **explain_setresgid_or_die** function is used to call the *setresgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setresgid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setresgid_on_error** function is used to call the *setresgid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setresgid*(3) function, but still returns to the caller.

*rgid* The rgid, exactly as to be passed to the *setresgid*(2) system call.

*egid* The egid, exactly as to be passed to the *setresgid*(2) system call.

*sgid* The sgid, exactly as to be passed to the *setresgid*(2) system call.

## RETURN VALUE

The **explain_setresgid_or_die** function only returns on success, see *setresgid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setresgid_on_error** function always returns the value return by the wrapped *setresgid*(2) system call.

## EXAMPLE

The **explain_setresgid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setresgid_or_die(rgid, egid, sgid);
```

## SEE ALSO

*setresgid*(2)
set real, effective and saved group ID

*explain_setresgid*(3)
explain *setresgid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setresuid − explain *setresuid*(2) errors

## SYNOPSIS

#include <libexplain/setresuid.h>

const char *explain_setresuid(uid_t ruid, uid_t euid, uid_t suid);
const char *explain_errno_setresuid(int errnum, uid_t ruid, uid_t euid, uid_t suid);
void explain_message_setresuid(char *message, int message_size, uid_t ruid, uid_t euid, uid_t suid);
void explain_message_errno_setresuid(char *message, int message_size, int errnum, uid_t ruid, uid_t euid, uid_t suid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setresuid*(2) system call.

### explain_setresuid

const char *explain_setresuid(uid_t ruid, uid_t euid, uid_t suid);

The **explain_setresuid** function is used to obtain an explanation of an error returned by the *setresuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*ruid* The original ruid, exactly as passed to the *setresuid*(2) system call.

*euid* The original euid, exactly as passed to the *setresuid*(2) system call.

*suid* The original suid, exactly as passed to the *setresuid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setresuid(ruid, euid, suid) < 0)
{
   fprintf(stderr, "%s\n", explain_setresuid(ruid, euid, suid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setresuid_or_die*(3) function.

### explain_errno_setresuid

const char *explain_errno_setresuid(int errnum, uid_t ruid, uid_t euid, uid_t suid);

The **explain_errno_setresuid** function is used to obtain an explanation of an error returned by the *setresuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ruid* The original ruid, exactly as passed to the *setresuid*(2) system call.

*euid* The original euid, exactly as passed to the *setresuid*(2) system call.

*suid* The original suid, exactly as passed to the *setresuid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setresuid(ruid, euid, suid) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setresuid(err, ruid,
   euid, suid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setresuid_or_die*(3) function.

**explain_message_setresuid**

void explain_message_setresuid(char *message, int message_size, uid_t ruid, uid_t euid, uid_t suid);

The **explain_message_setresuid** function is used to obtain an explanation of an error returned by the *setresuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*ruid* The original ruid, exactly as passed to the *setresuid*(2) system call.

*euid* The original euid, exactly as passed to the *setresuid*(2) system call.

*suid* The original suid, exactly as passed to the *setresuid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setresuid(ruid, euid, suid) < 0)
{
    char message[3000];
   explain_message_setresuid(message, sizeof(message), ruid,
   euid, suid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setresuid_or_die*(3) function.

**explain_message_errno_setresuid**

void explain_message_errno_setresuid(char *message, int message_size, int errnum, uid_t ruid, uid_t euid, uid_t suid);

The **explain_message_errno_setresuid** function is used to obtain an explanation of an error returned by the *setresuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ruid* The original ruid, exactly as passed to the *setresuid*(2) system call.

*euid* The original euid, exactly as passed to the *setresuid*(2) system call.

*suid* The original suid, exactly as passed to the *setresuid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setresuid(ruid, euid, suid) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setresuid(message, sizeof(message), err,
   ruid, euid, suid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setresuid_or_die*(3) function.

## SEE ALSO

*setresuid*(2)
set real, effective and saved user ID

*explain_setresuid_or_die*(3)
set real, effective and saved user ID and report errors

## COPYRIGHT

## NAME

explain_setresuid_or_die − set r/e/s user ID and report errors

## SYNOPSIS

#include <libexplain/setresuid.h>

void explain_setresuid_or_die(uid_t ruid, uid_t euid, uid_t suid);
int explain_setresuid_on_error(uid_t ruid, uid_t euid, uid_t suid);

## DESCRIPTION

The **explain_setresuid_or_die** function is used to call the *setresuid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setresuid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setresuid_on_error** function is used to call the *setresuid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setresuid*(3) function, but still returns to the caller.

*ruid* The ruid, exactly as to be passed to the *setresuid*(2) system call.

*euid* The euid, exactly as to be passed to the *setresuid*(2) system call.

*suid* The suid, exactly as to be passed to the *setresuid*(2) system call.

## RETURN VALUE

The **explain_setresuid_or_die** function only returns on success, see *setresuid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setresuid_on_error** function always returns the value return by the wrapped *setresuid*(2) system call.

## EXAMPLE

The **explain_setresuid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setresuid_or_die(ruid, euid, suid);
```

## SEE ALSO

*setresuid*(2)
set real, effective and saved user ID

*explain_setresuid*(3)
explain *setresuid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setreuid − explain *setreuid*(2) errors

## SYNOPSIS

#include <libexplain/setreuid.h>

const char *explain_setreuid(uid_t ruid, uid_t euid);
const char *explain_errno_setreuid(int errnum, uid_t ruid, uid_t euid);
void explain_message_setreuid(char *message, int message_size, uid_t ruid, uid_t euid);
void explain_message_errno_setreuid(char *message, int message_size, int errnum, uid_t ruid, uid_t euid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setreuid*(2) system call.

### explain_setreuid

const char *explain_setreuid(uid_t ruid, uid_t euid);

The **explain_setreuid** function is used to obtain an explanation of an error returned by the *setreuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*ruid* The original ruid, exactly as passed to the *setreuid*(2) system call.

*euid* The original euid, exactly as passed to the *setreuid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setreuid(ruid, euid) < 0)
{
   fprintf(stderr, "%s\n", explain_setreuid(ruid, euid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setreuid_or_die*(3) function.

### explain_errno_setreuid

const char *explain_errno_setreuid(int errnum, uid_t ruid, uid_t euid);

The **explain_errno_setreuid** function is used to obtain an explanation of an error returned by the *setreuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ruid* The original ruid, exactly as passed to the *setreuid*(2) system call.

*euid* The original euid, exactly as passed to the *setreuid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setreuid(ruid, euid) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_setreuid(err, ruid,
    euid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setreuid_or_die*(3) function.

**explain_message_setreuid**

void explain_message_setreuid(char *message, int message_size, uid_t ruid, uid_t euid);

The **explain_message_setreuid** function is used to obtain an explanation of an error returned by the *setreuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*ruid* The original ruid, exactly as passed to the *setreuid*(2) system call.

*euid* The original euid, exactly as passed to the *setreuid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setreuid(ruid, euid) < 0)
{
    char message[3000];
    explain_message_setreuid(message, sizeof(message), ruid,
    euid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setreuid_or_die*(3) function.

**explain_message_errno_setreuid**

void explain_message_errno_setreuid(char *message, int message_size, int errnum, uid_t ruid, uid_t euid);

The **explain_message_errno_setreuid** function is used to obtain an explanation of an error returned by the *setreuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*ruid* The original ruid, exactly as passed to the *setreuid*(2) system call.

*euid* The original euid, exactly as passed to the *setreuid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setreuid(ruid, euid) < 0)
{
```

```
    int err = errno;
    char message[3000];
    explain_message_errno_setreuid(message, sizeof(message), err,
    ruid, euid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setreuid_or_die*(3) function.

## SEE ALSO

*setreuid*(2)
: set the real and effective user ID

*explain_setreuid_or_die*(3)
: set the real and effective user ID and report errors

## COPYRIGHT

## NAME

explain_setreuid_or_die − set the real and effective user ID and report errors

## SYNOPSIS

#include <libexplain/setreuid.h>

void explain_setreuid_or_die(uid_t ruid, uid_t euid);
int explain_setreuid_on_error(uid_t ruid, uid_t euid);

## DESCRIPTION

The **explain_setreuid_or_die** function is used to call the *setreuid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setreuid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setreuid_on_error** function is used to call the *setreuid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setreuid*(3) function, but still returns to the caller.

*ruid* The ruid, exactly as to be passed to the *setreuid*(2) system call.

*euid* The euid, exactly as to be passed to the *setreuid*(2) system call.

## RETURN VALUE

The **explain_setreuid_or_die** function only returns on success, see *setreuid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setreuid_on_error** function always returns the value return by the wrapped *setreuid*(2) system call.

## EXAMPLE

The **explain_setreuid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setreuid_or_die(ruid, euid);
```

## SEE ALSO

*setreuid*(2)
set the real and effective user ID

*explain_setreuid*(3)
explain *setreuid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setsid − explain *setsid*(2) errors

## SYNOPSIS

#include <libexplain/setsid.h>

const char *explain_setsid(void);
const char *explain_errno_setsid(int errnum, void);
void explain_message_setsid(char *message, int message_size, void);
void explain_message_errno_setsid(char *message, int message_size, int errnum, void);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setsid*(2) system call.

### explain_setsid

const char *explain_setsid(void);

The **explain_setsid** function is used to obtain an explanation of an error returned by the *setsid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = setsid();
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_setsid());
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setsid_or_die*(3) function.

### explain_errno_setsid

const char *explain_errno_setsid(int errnum, void);

The **explain_errno_setsid** function is used to obtain an explanation of an error returned by the *setsid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = setsid();
if (result < 0)
{
    int err = errno;
```

```
    fprintf(stderr, "%s\n", explain_errno_setsid(err, ));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setsid_or_die*(3) function.

**explain_message_setsid**

void explain_message_setsid(char *message, int message_size, void);

The **explain_message_setsid** function is used to obtain an explanation of an error returned by the *setsid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = setsid();
if (result < 0)
{
    char message[3000];
    explain_message_setsid(message, sizeof(message), );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setsid_or_die*(3) function.

**explain_message_errno_setsid**

void explain_message_errno_setsid(char *message, int message_size, int errnum, void);

The **explain_message_errno_setsid** function is used to obtain an explanation of an error returned by the *setsid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = setsid();
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_setsid(message, sizeof(message), err, );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setsid_or_die*(3) function.

## SEE ALSO

*setsid*(2) creates a session and sets the process group ID

*explain_setsid_or_die*(3)
creates a session and sets the process group ID and report errors

## COPYRIGHT

## NAME

explain_setsid_or_die − sets process group ID and report errors

## SYNOPSIS

#include <libexplain/setsid.h>

pid_t explain_setsid_or_die(void);
pid_t explain_setsid_on_error(void);

## DESCRIPTION

The **explain_setsid_or_die** function is used to call the *setsid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setsid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setsid_on_error** function is used to call the *setsid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setsid*(3) function, but still returns to the caller.

## RETURN VALUE

The **explain_setsid_or_die** function only returns on success, see *setsid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setsid_on_error** function always returns the value return by the wrapped *setsid*(2) system call.

## EXAMPLE

The **explain_setsid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setsid_or_die();
```

## SEE ALSO

*setsid*(2) creates a session and sets the process group ID

*explain_setsid*(3)
explain *setsid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setsockopt − explain setsockopt(2) errors

## SYNOPSIS

#include <libexplain/setsockopt.h>

const char *explain_setsockopt(int fildes, int level, int name, void *data, socklen_t data_size);
const char *explain_errno_setsockopt(int errnum, int fildes, int level, int name, void *data, socklen_t data_size);
void explain_message_setsockopt(char *message, int message_size, int fildes, int level, int name, void *data, socklen_t data_size);
void explain_message_errno_setsockopt(char *message, int message_size, int errnum, int fildes, int level, int name, void *data, socklen_t data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setsockopt*(2) system call.

### explain_setsockopt

const char *explain_setsockopt(int fildes, int level, int name, void *data, socklen_t data_size);

The **explain_setsockopt** function is used to obtain an explanation of an error returned by the *setsockopt*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (setsockopt(fildes, level, name, data, data_size) < 0)
{
    fprintf(stderr, "%s\n", explain_setsockopt(fildes,
        level, name, data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setsockopt_or_die*(3) function.

*fildes* The original fildes, exactly as passed to the *setsockopt*(2) system call.

*level* The original level, exactly as passed to the *setsockopt*(2) system call.

*name* The original name, exactly as passed to the *setsockopt*(2) system call.

*data* The original data, exactly as passed to the *setsockopt*(2) system call.

*data_size*
The original data_size, exactly as passed to the *setsockopt*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_setsockopt

const char *explain_errno_setsockopt(int errnum, int fildes, int level, int name, void *data, socklen_t data_size);

The **explain_errno_setsockopt** function is used to obtain an explanation of an error returned by the *setsockopt*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (setsockopt(fildes, level, name, data, data_size) < 0)
```

```
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_setsockopt(err,
        fildes, level, name, data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setsockopt_or_die*(3) function.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *setsockopt*(2) system call.

*level* The original level, exactly as passed to the *setsockopt*(2) system call.

*name* The original name, exactly as passed to the *setsockopt*(2) system call.

*data* The original data, exactly as passed to the *setsockopt*(2) system call.

*data_size*
The original data_size, exactly as passed to the *setsockopt*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_setsockopt**

void explain_message_setsockopt(char *message, int message_size, int fildes, int level, int name, void *data, socklen_t data_size);

The **explain_message_setsockopt** function may be used to obtain an explanation of an error returned by the *setsockopt*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (setsockopt(fildes, level, name, data, data_size) < 0)
{
    char message[3000];
    explain_message_setsockopt(message, sizeof(message),
        fildes, level, name, data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setsockopt_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *setsockopt*(2) system call.

*level* The original level, exactly as passed to the *setsockopt*(2) system call.

*name* The original name, exactly as passed to the *setsockopt*(2) system call.

*data* The original data, exactly as passed to the *setsockopt*(2) system call.

*data_size*
The original data_size, exactly as passed to the *setsockopt*(2) system call.

**explain_message_errno_setsockopt**

void explain_message_errno_setsockopt(char *message, int message_size, int errnum, int fildes, int level, int name, void *data, socklen_t data_size);

The **explain_message_errno_setsockopt** function may be used to obtain an explanation of an error returned by the *setsockopt*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (setsockopt(fildes, level, name, data, data_size) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_setsockopt(message, sizeof(message),
        err, fildes, level, name, data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setsockopt_or_die*(3) function.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *setsockopt*(2) system call.

*level* The original level, exactly as passed to the *setsockopt*(2) system call.

*name* The original name, exactly as passed to the *setsockopt*(2) system call.

*data* The original data, exactly as passed to the *setsockopt*(2) system call.

*data_size*
The original data_size, exactly as passed to the *setsockopt*(2) system call.

## SEE ALSO

*setsockopt*(2)
get and set options on sockets

*explain_setsockopt_or_die*(3)
get and set options on sockets and report errors

## COPYRIGHT

## NAME

explain_setsockopt_or_die − get and set options on sockets and report errors

## SYNOPSIS

#include <libexplain/setsockopt.h>

void explain_setsockopt_or_die(int fildes, int level, int name, void *data, socklen_t data_size);

## DESCRIPTION

The **explain_setsockopt_or_die** function is used to call the *setsockopt*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_setsockopt*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_setsockopt_or_die(fildes, level, name, data, data_size);
```

*fildes* The fildes, exactly as to be passed to the *setsockopt*(2) system call.

*level* The level, exactly as to be passed to the *setsockopt*(2) system call.

*name* The name, exactly as to be passed to the *setsockopt*(2) system call.

*data* The data, exactly as to be passed to the *setsockopt*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *setsockopt*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*setsockopt*(2)
get and set options on sockets

*explain_setsockopt*(3)
explain *setsockopt*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setuid − explain *setuid*(2) errors

## SYNOPSIS

#include <libexplain/setuid.h>

const char *explain_setuid(int uid);
const char *explain_errno_setuid(int errnum, int uid);
void explain_message_setuid(char *message, int message_size, int uid);
void explain_message_errno_setuid(char *message, int message_size, int errnum, int uid);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setuid*(2) system call.

### explain_setuid

const char *explain_setuid(int uid);

The **explain_setuid** function is used to obtain an explanation of an error returned by the *setuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*uid* The original uid, exactly as passed to the *setuid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setuid(uid) < 0)
{
   fprintf(stderr, "%s\n", explain_setuid(uid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setuid_or_die*(3) function.

### explain_errno_setuid

const char *explain_errno_setuid(int errnum, int uid);

The **explain_errno_setuid** function is used to obtain an explanation of an error returned by the *setuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*uid* The original uid, exactly as passed to the *setuid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setuid(uid) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_setuid(err, uid));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setuid_or_die*(3) function.

**explain_message_setuid**

void explain_message_setuid(char *message, int message_size, int uid);

The **explain_message_setuid** function is used to obtain an explanation of an error returned by the *setuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*uid* The original uid, exactly as passed to the *setuid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setuid(uid) < 0)
{
    char message[3000];
    explain_message_setuid(message, sizeof(message), uid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setuid_or_die*(3) function.

**explain_message_errno_setuid**

void explain_message_errno_setuid(char *message, int message_size, int errnum, int uid);

The **explain_message_errno_setuid** function is used to obtain an explanation of an error returned by the *setuid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*uid* The original uid, exactly as passed to the *setuid*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setuid(uid) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_setuid(message, sizeof(message), err,
    uid);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_setuid_or_die*(3) function.

## SEE ALSO

*setuid*(2)
set user identity

*explain_setuid_or_die*(3)
set user identity and report errors

## COPYRIGHT

## NAME

explain_setuid_or_die − set user identity and report errors

## SYNOPSIS

#include <libexplain/setuid.h>

void explain_setuid_or_die(int uid);
int explain_setuid_on_error(int uid);

## DESCRIPTION

The **explain_setuid_or_die** function is used to call the *setuid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setuid*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setuid_on_error** function is used to call the *setuid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setuid*(3) function, but still returns to the caller.

*uid* The uid, exactly as to be passed to the *setuid*(2) system call.

## RETURN VALUE

The **explain_setuid_or_die** function only returns on success, see *setuid*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setuid_on_error** function always returns the value return by the wrapped *setuid*(2) system call.

## EXAMPLE

The **explain_setuid_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setuid_or_die(uid);
```

## SEE ALSO

*setuid*(2)
set user identity

*explain_setuid*(3)
explain *setuid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_setvbuf − explain *setvbuf*(3) errors

## SYNOPSIS

#include <libexplain/setvbuf.h>

const char *explain_setvbuf(FILE *fp, char *data, int mode, size_t size);
const char *explain_errno_setvbuf(int errnum, FILE *fp, char *data, int mode, size_t size);
void explain_message_setvbuf(char *message, int message_size, FILE *fp, char *data, int mode, size_t size);
void explain_message_errno_setvbuf(char *message, int message_size, int errnum, FILE *fp, char *data, int mode, size_t size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *setvbuf*(3) system call.

### explain_setvbuf

const char *explain_setvbuf(FILE *fp, char *data, int mode, size_t size);

The **explain_setvbuf** function is used to obtain an explanation of an error returned by the *setvbuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *setvbuf*(3) system call.

*data* The original data, exactly as passed to the *setvbuf*(3) system call.

*mode* The original mode, exactly as passed to the *setvbuf*(3) system call.

*size* The original size, exactly as passed to the *setvbuf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setvbuf(fp, data, mode, size) < 0)
{
    fprintf(stderr, "%s\n", explain_setvbuf(fp, data, mode,
    size));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setvbuf_or_die*(3) function.

### explain_errno_setvbuf

const char *explain_errno_setvbuf(int errnum, FILE *fp, char *data, int mode, size_t size);

The **explain_errno_setvbuf** function is used to obtain an explanation of an error returned by the *setvbuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *setvbuf*(3) system call.

*data* The original data, exactly as passed to the *setvbuf*(3) system call.

*mode* The original mode, exactly as passed to the *setvbuf*(3) system call.

*size* The original size, exactly as passed to the *setvbuf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setvbuf(fp, data, mode, size) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_setvbuf(err, fp, data,
   mode, size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setvbuf_or_die*(3) function.

**explain_message_setvbuf**

void explain_message_setvbuf(char *message, int message_size, FILE *fp, char *data, int mode, size_t size);

The **explain_message_setvbuf** function is used to obtain an explanation of an error returned by the *setvbuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *setvbuf*(3) system call.

*data* The original data, exactly as passed to the *setvbuf*(3) system call.

*mode* The original mode, exactly as passed to the *setvbuf*(3) system call.

*size* The original size, exactly as passed to the *setvbuf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setvbuf(fp, data, mode, size) < 0)
{
    char message[3000];
   explain_message_setvbuf(message, sizeof(message), fp, data,
   mode, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setvbuf_or_die*(3) function.

**explain_message_errno_setvbuf**

void explain_message_errno_setvbuf(char *message, int message_size, int errnum, FILE *fp, char *data, int mode, size_t size);

The **explain_message_errno_setvbuf** function is used to obtain an explanation of an error returned by the *setvbuf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *setvbuf*(3) system call.

*data* The original data, exactly as passed to the *setvbuf*(3) system call.

*mode* The original mode, exactly as passed to the *setvbuf*(3) system call.

*size* The original size, exactly as passed to the *setvbuf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (setvbuf(fp, data, mode, size) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_setvbuf(message, sizeof(message), err,
   fp, data, mode, size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_setvbuf_or_die*(3) function.

## SEE ALSO

*setvbuf*(3)
stream buffering operations

*explain_setvbuf_or_die*(3)
stream buffering operations and report errors

## COPYRIGHT

## NAME

explain_setvbuf_or_die − stream buffering operations and report errors

## SYNOPSIS

#include <libexplain/setvbuf.h>

void explain_setvbuf_or_die(FILE *fp, char *data, int mode, size_t size);
int explain_setvbuf_on_error(FILE *fp, char *data, int mode, size_t size);

## DESCRIPTION

The **explain_setvbuf_or_die** function is used to call the *setvbuf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setvbuf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_setvbuf_on_error** function is used to call the *setvbuf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_setvbuf*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *setvbuf*(3) system call.

*data* The data, exactly as to be passed to the *setvbuf*(3) system call.

*mode* The mode, exactly as to be passed to the *setvbuf*(3) system call.

*size* The size, exactly as to be passed to the *setvbuf*(3) system call.

## RETURN VALUE

The **explain_setvbuf_or_die** function only returns on success, see *setvbuf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_setvbuf_on_error** function always returns the value return by the wrapped *setvbuf*(3) system call.

## EXAMPLE

The **explain_setvbuf_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_setvbuf_or_die(fp, data, mode, size);
```

## SEE ALSO

*setvbuf*(3)
stream buffering operations

*explain_setvbuf*(3)
explain *setvbuf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_shmat − explain *shmat*(2) errors

## SYNOPSIS

#include <libexplain/shmat.h>

const char *explain_shmat(int shmid, const void *shmaddr, int shmflg);
const char *explain_errno_shmat(int errnum, int shmid, const void *shmaddr, int shmflg);
void explain_message_shmat(char *message, int message_size, int shmid, const void *shmaddr, int shmflg);
void explain_message_errno_shmat(char *message, int message_size, int errnum, int shmid, const void *shmaddr, int shmflg);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *shmat*(2) system call.

### explain_shmat

const char *explain_shmat(int shmid, const void *shmaddr, int shmflg);

The **explain_shmat** function is used to obtain an explanation of an error returned by the *shmat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*shmid* The original shmid, exactly as passed to the *shmat*(2) system call.

*shmaddr* The original shmaddr, exactly as passed to the *shmat*(2) system call.

*shmflg* The original shmflg, exactly as passed to the *shmat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void *result = shmat(shmid, shmaddr, shmflg);
if (!result)
{
    fprintf(stderr, "%s\n", explain_shmat(shmid, shmaddr,
    shmflg));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_shmat_or_die*(3) function.

### explain_errno_shmat

const char *explain_errno_shmat(int errnum, int shmid, const void *shmaddr, int shmflg);

The **explain_errno_shmat** function is used to obtain an explanation of an error returned by the *shmat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*shmid* The original shmid, exactly as passed to the *shmat*(2) system call.

*shmaddr* The original shmaddr, exactly as passed to the *shmat*(2) system call.

*shmflg* The original shmflg, exactly as passed to the *shmat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void *result = shmat(shmid, shmaddr, shmflg);
if (!result)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_shmat(err, shmid,
   shmaddr, shmflg));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_shmat_or_die*(3) function.

**explain_message_shmat**

void explain_message_shmat(char *message, int message_size, int shmid, const void *shmaddr, int shmflg);

The **explain_message_shmat** function is used to obtain an explanation of an error returned by the *shmat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*shmid* The original shmid, exactly as passed to the *shmat*(2) system call.

*shmaddr* The original shmaddr, exactly as passed to the *shmat*(2) system call.

*shmflg* The original shmflg, exactly as passed to the *shmat*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void *result = shmat(shmid, shmaddr, shmflg);
if (!result)
{
    char message[3000];
   explain_message_shmat(message, sizeof(message), shmid,
   shmaddr, shmflg);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_shmat_or_die*(3) function.

**explain_message_errno_shmat**

void explain_message_errno_shmat(char *message, int message_size, int errnum, int shmid, const void *shmaddr, int shmflg);

The **explain_message_errno_shmat** function is used to obtain an explanation of an error returned by the *shmat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*shmid* The original shmid, exactly as passed to the *shmat*(2) system call.

*shmaddr* The original shmaddr, exactly as passed to the *shmat*(2) system call.

*shmflg* The original shmflg, exactly as passed to the *shmat*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
void *result = shmat(shmid, shmaddr, shmflg);
if (!result)
{
    int err = errno;
    char message[3000];
   explain_message_errno_shmat(message, sizeof(message), err,
   shmid, shmaddr, shmflg);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_shmat_or_die*(3) function.

## SEE ALSO

*shmat*(2)
shared memory attach

*explain_shmat_or_die*(3)
shared memory attach and report errors

## COPYRIGHT

## NAME

explain_shmat_or_die − shared memory attach and report errors

## SYNOPSIS

#include <libexplain/shmat.h>

void *explain_shmat_or_die(int shmid, const void *shmaddr, int shmflg);
void *explain_shmat_on_error(int shmid, const void *shmaddr, int shmflg);

## DESCRIPTION

The **explain_shmat_or_die** function is used to call the *shmat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_shmat*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_shmat_on_error** function is used to call the *shmat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_shmat*(3) function, but still returns to the caller.

*shmid* The shmid, exactly as to be passed to the *shmat*(2) system call.

*shmaddr* The shmaddr, exactly as to be passed to the *shmat*(2) system call.

*shmflg* The shmflg, exactly as to be passed to the *shmat*(2) system call.

## RETURN VALUE

The **explain_shmat_or_die** function only returns on success, see *shmat*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_shmat_on_error** function always returns the value return by the wrapped *shmat*(2) system call.

## EXAMPLE

The **explain_shmat_or_die** function is intended to be used in a fashion similar to the following example:

```
void *result = explain_shmat_or_die(shmid, shmaddr, shmflg);
```

## SEE ALSO

*shmat*(2)
shared memory attach

*explain_shmat*(3)
explain *shmat*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_shmctl − explain *shmctl*(2) errors

## SYNOPSIS

#include <libexplain/shmctl.h>

const char *explain_shmctl(int shmid, int command, struct shmid_ds *data);
const char *explain_errno_shmctl(int errnum, int shmid, int command, struct shmid_ds *data);
void explain_message_shmctl(char *message, int message_size, int shmid, int command, struct shmid_ds *data);
void explain_message_errno_shmctl(char *message, int message_size, int errnum, int shmid, int command, struct shmid_ds *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *shmctl*(2) system call.

### explain_shmctl

const char *explain_shmctl(int shmid, int command, struct shmid_ds *data);

The **explain_shmctl** function is used to obtain an explanation of an error returned by the *shmctl*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*shmid* The original shmid, exactly as passed to the *shmctl*(2) system call.

*command*
The original command, exactly as passed to the *shmctl*(2) system call.

*data* The original data, exactly as passed to the *shmctl*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (shmctl(shmid, command, data) < 0)
{
    fprintf(stderr, "%s\n", explain_shmctl(shmid, command, data));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_shmctl_or_die*(3) function.

### explain_errno_shmctl

const char *explain_errno_shmctl(int errnum, int shmid, int command, struct shmid_ds *data);

The **explain_errno_shmctl** function is used to obtain an explanation of an error returned by the *shmctl*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*shmid* The original shmid, exactly as passed to the *shmctl*(2) system call.

*command*
The original command, exactly as passed to the *shmctl*(2) system call.

*data* The original data, exactly as passed to the *shmctl*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (shmctl(shmid, command, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_shmctl(err, shmid,
   command, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_shmctl_or_die*(3) function.

**explain_message_shmctl**

void explain_message_shmctl(char *message, int message_size, int shmid, int command, struct shmid_ds *data);

The **explain_message_shmctl** function is used to obtain an explanation of an error returned by the *shmctl*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*shmid* The original shmid, exactly as passed to the *shmctl*(2) system call.

*command*
The original command, exactly as passed to the *shmctl*(2) system call.

*data* The original data, exactly as passed to the *shmctl*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (shmctl(shmid, command, data) < 0)
{
    char message[3000];
   explain_message_shmctl(message, sizeof(message), shmid,
   command, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_shmctl_or_die*(3) function.

**explain_message_errno_shmctl**

void explain_message_errno_shmctl(char *message, int message_size, int errnum, int shmid, int command, struct shmid_ds *data);

The **explain_message_errno_shmctl** function is used to obtain an explanation of an error returned by the *shmctl*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*shmid* The original shmid, exactly as passed to the *shmctl*(2) system call.

*command*
The original command, exactly as passed to the *shmctl*(2) system call.

*data* The original data, exactly as passed to the *shmctl*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (shmctl(shmid, command, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_shmctl(message, sizeof(message), err,
   shmid, command, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_shmctl_or_die*(3) function.

## SEE ALSO

*shmctl*(2)
shared memory control

*explain_shmctl_or_die*(3)
shared memory control and report errors

## COPYRIGHT

## NAME

explain_shmctl_or_die − shared memory control and report errors

## SYNOPSIS

#include <libexplain/shmctl.h>

void explain_shmctl_or_die(int shmid, int command, struct shmid_ds *data);
int explain_shmctl_on_error(int shmid, int command, struct shmid_ds *data);

## DESCRIPTION

The **explain_shmctl_or_die** function is used to call the *shmctl*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_shmctl*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_shmctl_on_error** function is used to call the *shmctl*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_shmctl*(3) function, but still returns to the caller.

*shmid* The shmid, exactly as to be passed to the *shmctl*(2) system call.

*command*
The command, exactly as to be passed to the *shmctl*(2) system call.

*data* The data, exactly as to be passed to the *shmctl*(2) system call.

## RETURN VALUE

The **explain_shmctl_or_die** function only returns on success, see *shmctl*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_shmctl_on_error** function always returns the value return by the wrapped *shmctl*(2) system call.

## EXAMPLE

The **explain_shmctl_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_shmctl_or_die(shmid, command, data);
```

## SEE ALSO

*shmctl*(2)
shared memory control

*explain_shmctl*(3)
explain *shmctl*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_signalfd − explain signalfd(2) errors

## SYNOPSIS

#include <libexplain/signalfd.h>

const char *explain_signalfd(int fildes, const sigset_t *mask, int flags);
const char *explain_errno_signalfd(int errnum, int fildes, const sigset_t *mask, int flags);
void explain_message_signalfd(char *message, int message_size, int fildes, const sigset_t *mask, int flags);
void explain_message_errno_signalfd(char *message, int message_size, int errnum, int fildes, const sigset_t *mask, int flags);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *signalfd*(2) system call.

### explain_signalfd

const char *explain_signalfd(int fildes, const sigset_t *mask, int flags);

The **explain_signalfd** function is used to obtain an explanation of an error returned by the *signalfd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *signalfd*(2) system call.

*mask* The original mask, exactly as passed to the *signalfd*(2) system call.

*flags* The original flags, exactly as passed to the *signalfd*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = signalfd(fildes, mask, flags);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_signalfd(fildes, mask,
   flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_signalfd_or_die*(3) function.

### explain_errno_signalfd

const char *explain_errno_signalfd(int errnum, int fildes, const sigset_t *mask, int flags);

The **explain_errno_signalfd** function is used to obtain an explanation of an error returned by the *signalfd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *signalfd*(2) system call.

*mask* The original mask, exactly as passed to the *signalfd*(2) system call.

*flags* The original flags, exactly as passed to the *signalfd*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = signalfd(fildes, mask, flags);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_signalfd(err, fildes,
   mask, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_signalfd_or_die*(3) function.

**explain_message_signalfd**

void explain_message_signalfd(char *message, int message_size, int fildes, const sigset_t *mask, int flags);

The **explain_message_signalfd** function is used to obtain an explanation of an error returned by the *signalfd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *signalfd*(2) system call.

*mask* The original mask, exactly as passed to the *signalfd*(2) system call.

*flags* The original flags, exactly as passed to the *signalfd*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = signalfd(fildes, mask, flags);
if (result < 0)
{
    char message[3000];
   explain_message_signalfd(message, sizeof(message), fildes,
   mask, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_signalfd_or_die*(3) function.

**explain_message_errno_signalfd**

void explain_message_errno_signalfd(char *message, int message_size, int errnum, int fildes, const sigset_t *mask, int flags);

The **explain_message_errno_signalfd** function is used to obtain an explanation of an error returned by the *signalfd*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes*
: The original fildes, exactly as passed to the *signalfd*(2) system call.

*mask*
: The original mask, exactly as passed to the *signalfd*(2) system call.

*flags*
: The original flags, exactly as passed to the *signalfd*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = signalfd(fildes, mask, flags);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_signalfd(message, sizeof(message), err,
   fildes, mask, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_signalfd_or_die*(3) function.

## SEE ALSO

*signalfd*(2)
: create a file descriptor for accepting signals

*explain_signalfd_or_die*(3)
: create a file descriptor for accepting signals and report errors

## COPYRIGHT

## NAME

explain_signalfd_or_die − create signalable file descriptor and report errors

## SYNOPSIS

#include <libexplain/signalfd.h>

int explain_signalfd_or_die(int fildes, const sigset_t *mask, int flags);
int explain_signalfd_on_error(int fildes, const sigset_t *mask, int flags);

## DESCRIPTION

The **explain_signalfd_or_die** function is used to call the *signalfd*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_signalfd*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_signalfd_on_error** function is used to call the *signalfd*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_signalfd*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *signalfd*(2) system call.

*mask* The mask, exactly as to be passed to the *signalfd*(2) system call.

*flags* The flags, exactly as to be passed to the *signalfd*(2) system call.

## RETURN VALUE

The **explain_signalfd_or_die** function only returns on success, see *signalfd*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_signalfd_on_error** function always returns the value return by the wrapped *signalfd*(2) system call.

## EXAMPLE

The **explain_signalfd_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_signalfd_or_die(fildes, mask, flags);
```

## SEE ALSO

*signalfd*(2)
create a file descriptor for accepting signals

*explain_signalfd*(3)
explain *signalfd*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_snprintf − explain *snprintf*(3) errors

## SYNOPSIS

#include <libexplain/snprintf.h>

const char *explain_snprintf(char *data, size_t data_size, const char *format);
const char *explain_errno_snprintf(int errnum, char *data, size_t data_size, const char *format);
void explain_message_snprintf(char *message, int message_size, char *data, size_t data_size, const char *format);
void explain_message_errno_snprintf(char *message, int message_size, int errnum, char *data, size_t data_size, const char *format);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *snprintf*(3) system call.

### explain_snprintf

const char *explain_snprintf(char *data, size_t data_size, const char *format);

The **explain_snprintf** function is used to obtain an explanation of an error returned by the *snprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *snprintf*(3) system call.

*data_size*
The original data_size, exactly as passed to the *snprintf*(3) system call.

*format* The original format, exactly as passed to the *snprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = snprintf(data, data_size, format);
if (result < 0 && errno != 0)
{
   fprintf(stderr, "%s\n", explain_snprintf(data, data_size,
   format));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_snprintf_or_die*(3) function.

### explain_errno_snprintf

const char *explain_errno_snprintf(int errnum, char *data, size_t data_size, const char *format);

The **explain_errno_snprintf** function is used to obtain an explanation of an error returned by the *snprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *snprintf*(3) system call.

*data_size*
The original data_size, exactly as passed to the *snprintf*(3) system call.

*format* The original format, exactly as passed to the *snprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = snprintf(data, data_size, format);
if (result < 0 && errno != 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_snprintf(err, data,
   data_size, format));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_snprintf_or_die*(3) function.

**explain_message_snprintf**

void explain_message_snprintf(char *message, int message_size, char *data, size_t data_size, const char *format);

The **explain_message_snprintf** function is used to obtain an explanation of an error returned by the *snprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *snprintf*(3) system call.

*data_size*
The original data_size, exactly as passed to the *snprintf*(3) system call.

*format* The original format, exactly as passed to the *snprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = snprintf(data, data_size, format);
if (result < 0 && errno != 0)
{
    char message[3000];
   explain_message_snprintf(message, sizeof(message), data,
   data_size, format);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_snprintf_or_die*(3) function.

**explain_message_errno_snprintf**

void explain_message_errno_snprintf(char *message, int message_size, int errnum, char *data, size_t data_size, const char *format);

The **explain_message_errno_snprintf** function is used to obtain an explanation of an error returned by the *snprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *snprintf*(3) system call.

*data_size*
The original data_size, exactly as passed to the *snprintf*(3) system call.

*format* The original format, exactly as passed to the *snprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = snprintf(data, data_size, format);
if (result < 0 && errno != 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_snprintf(message, sizeof(message), err,
   data, data_size, format);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_snprintf_or_die*(3) function.

## SEE ALSO

*snprintf*(3)
formatted output conversion

*explain_snprintf_or_die*(3)
formatted output conversion and report errors

## COPYRIGHT

## NAME

explain_snprintf_or_die − formatted output conversion and report errors

## SYNOPSIS

#include <libexplain/snprintf.h>

int explain_snprintf_or_die(char *data, size_t data_size, const char *format);
int explain_snprintf_on_error(char *data, size_t data_size, const char *format);

## DESCRIPTION

The **explain_snprintf_or_die** function is used to call the *snprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_snprintf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_snprintf_on_error** function is used to call the *snprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_snprintf*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *snprintf*(3) system call.

*data_size*
The data_size, exactly as to be passed to the *snprintf*(3) system call.

*format* The format, exactly as to be passed to the *snprintf*(3) system call.

## RETURN VALUE

The **explain_snprintf_or_die** function only returns on success, see *snprintf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_snprintf_on_error** function always returns the value return by the wrapped *snprintf*(3) system call.

## EXAMPLE

The **explain_snprintf_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_snprintf_or_die(data, data_size, format);
```

## SEE ALSO

*snprintf*(3)
formatted output conversion

*explain_snprintf*(3)
explain *snprintf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_socket − explain socket(2) errors

## SYNOPSIS

#include <libexplain/socket.h>

const char *explain_socket(int domain, int type, int protocol);
const char *explain_errno_socket(int errnum, int domain, int type, int protocol);
void explain_message_socket(char *message, int message_size, int domain, int type, int protocol);
void explain_message_errno_socket(char *message, int message_size, int errnum, int domain, int type, int protocol);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *socket*(2) system call.

### explain_socket

const char *explain_socket(int domain, int type, int protocol);

The **explain_socket** function is used to obtain an explanation of an error returned by the *socket*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (socket(domain, type, protocol) < 0)
{
    fprintf(stderr, "%s\n", explain_socket(domain, type, protocol));
    exit(EXIT_FAILURE);
}
```

*domain* The original domain, exactly as passed to the *socket*(2) system call.

*type* The original type, exactly as passed to the *socket*(2) system call.

*protocol* The original protocol, exactly as passed to the *socket*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_socket

const char *explain_errno_socket(int errnum, int domain, int type, int protocol);

The **explain_errno_socket** function is used to obtain an explanation of an error returned by the *socket*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (socket(domain, type, protocol) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_socket(err,
        domain, type, protocol));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*domain* The original domain, exactly as passed to the *socket*(2) system call.

*type* The original type, exactly as passed to the *socket*(2) system call.

*protocol* The original protocol, exactly as passed to the *socket*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_socket**

void explain_message_socket(char *message, int message_size, int domain, int type, int protocol);

The **explain_message_socket** function may be used to obtain an explanation of an error returned by the *socket*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (socket(domain, type, protocol) < 0)
{
    char message[3000];
    explain_message_socket(message, sizeof(message), domain, type, protocol);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*domain* The original domain, exactly as passed to the *socket*(2) system call.

*type* The original type, exactly as passed to the *socket*(2) system call.

*protocol* The original protocol, exactly as passed to the *socket*(2) system call.

**explain_message_errno_socket**

void explain_message_errno_socket(char *message, int message_size, int errnum, int domain, int type, int protocol);

The **explain_message_errno_socket** function may be used to obtain an explanation of an error returned by the *socket*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (socket(domain, type, protocol) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_socket(message, sizeof(message), err,
        domain, type, protocol);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*domain*
: The original domain, exactly as passed to the *socket*(2) system call.

*type*
: The original type, exactly as passed to the *socket*(2) system call.

*protocol*
: The original protocol, exactly as passed to the *socket*(2) system call.

## SEE ALSO

*socket*(2)
: create an endpoint for communication

*explain_socket_or_die*(3)
: create an endpoint for communication and report errors

## COPYRIGHT

## NAME

explain_socket_or_die − create an endpoint and report errors

## SYNOPSIS

#include <libexplain/socket.h>

void explain_socket_or_die(int domain, int type, int protocol);

## DESCRIPTION

The **explain_socket_or_die** function is used to call the *socket*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_socket*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_socket_or_die(domain, type, protocol);
```

*domain* The domain, exactly as to be passed to the *socket*(2) system call.

*type* The type, exactly as to be passed to the *socket*(2) system call.

*protocol* The protocol, exactly as to be passed to the *socket*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*socket*(2)
create an endpoint for communication

*explain_socket*(3)
explain *socket*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_socketpair − explain *socketpair*(2) errors

## SYNOPSIS

#include <libexplain/socketpair.h>

const char *explain_socketpair(int domain, int type, int protocol, int *sv);
const char *explain_errno_socketpair(int errnum, int domain, int type, int protocol, int *sv);
void explain_message_socketpair(char *message, int message_size, int domain, int type, int protocol, int *sv);
void explain_message_errno_socketpair(char *message, int message_size, int errnum, int domain, int type, int protocol, int *sv);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *socketpair*(2) system call.

### explain_socketpair

const char *explain_socketpair(int domain, int type, int protocol, int *sv);

The **explain_socketpair** function is used to obtain an explanation of an error returned by the *socketpair*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*domain* The original domain, exactly as passed to the *socketpair*(2) system call.

*type* The original type, exactly as passed to the *socketpair*(2) system call.

*protocol* The original protocol, exactly as passed to the *socketpair*(2) system call.

*sv* The original sv, exactly as passed to the *socketpair*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (socketpair(domain, type, protocol, sv) < 0)
{
   fprintf(stderr, "%s\n", explain_socketpair(domain, type,
   protocol, sv));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_socketpair_or_die*(3) function.

### explain_errno_socketpair

const char *explain_errno_socketpair(int errnum, int domain, int type, int protocol, int *sv);

The **explain_errno_socketpair** function is used to obtain an explanation of an error returned by the *socketpair*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*domain* The original domain, exactly as passed to the *socketpair*(2) system call.

*type* The original type, exactly as passed to the *socketpair*(2) system call.

*protocol* The original protocol, exactly as passed to the *socketpair*(2) system call.

*sv* The original sv, exactly as passed to the *socketpair*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (socketpair(domain, type, protocol, sv) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_socketpair(err, domain,
   type, protocol, sv));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_socketpair_or_die*(3) function.

**explain_message_socketpair**

void explain_message_socketpair(char *message, int message_size, int domain, int type, int protocol, int *sv);

The **explain_message_socketpair** function is used to obtain an explanation of an error returned by the *socketpair*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*domain* The original domain, exactly as passed to the *socketpair*(2) system call.

*type* The original type, exactly as passed to the *socketpair*(2) system call.

*protocol* The original protocol, exactly as passed to the *socketpair*(2) system call.

*sv* The original sv, exactly as passed to the *socketpair*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (socketpair(domain, type, protocol, sv) < 0)
{
    char message[3000];
   explain_message_socketpair(message, sizeof(message), domain,
   type, protocol, sv);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_socketpair_or_die*(3) function.

**explain_message_errno_socketpair**

void explain_message_errno_socketpair(char *message, int message_size, int errnum, int domain, int type, int protocol, int *sv);

The **explain_message_errno_socketpair** function is used to obtain an explanation of an error returned by the *socketpair*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*domain* The original domain, exactly as passed to the *socketpair*(2) system call.

*type* The original type, exactly as passed to the *socketpair*(2) system call.

*protocol* The original protocol, exactly as passed to the *socketpair*(2) system call.

*sv* The original sv, exactly as passed to the *socketpair*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (socketpair(domain, type, protocol, sv) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_socketpair(message, sizeof(message),
   err, domain, type, protocol, sv);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_socketpair_or_die*(3) function.

## SEE ALSO

*socketpair*(2)
create a pair of connected sockets

*explain_socketpair_or_die*(3)
create a pair of connected sockets and report errors

## COPYRIGHT

## NAME

explain_socketpair_or_die − create pair of connected sockets and report errors

## SYNOPSIS

#include <libexplain/socketpair.h>

void explain_socketpair_or_die(int domain, int type, int protocol, int *sv);
int explain_socketpair_on_error(int domain, int type, int protocol, int *sv);

## DESCRIPTION

The **explain_socketpair_or_die** function is used to call the *socketpair*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_socketpair*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_socketpair_on_error** function is used to call the *socketpair*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_socketpair*(3) function, but still returns to the caller.

*domain* The domain, exactly as to be passed to the *socketpair*(2) system call.

*type* The type, exactly as to be passed to the *socketpair*(2) system call.

*protocol* The protocol, exactly as to be passed to the *socketpair*(2) system call.

*sv* The sv, exactly as to be passed to the *socketpair*(2) system call.

## RETURN VALUE

The **explain_socketpair_or_die** function only returns on success, see *socketpair*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_socketpair_on_error** function always returns the value return by the wrapped *socketpair*(2) system call.

## EXAMPLE

The **explain_socketpair_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_socketpair_or_die(domain, type, protocol, sv);
```

## SEE ALSO

*socketpair*(2)
create a pair of connected sockets

*explain_socketpair*(3)
explain *socketpair*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_sprintf − explain *sprintf*(3) errors

## SYNOPSIS

#include <libexplain/sprintf.h>

const char *explain_sprintf(char *data, const char *format, ...);
const char *explain_errno_sprintf(int errnum, char *data, const char *format, ...);
void explain_message_sprintf(char *message, int message_size, char *data, const char *format, ...);
void explain_message_errno_sprintf(char *message, int message_size, int errnum, char *data, const char *format, ...);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *sprintf*(3) system call.

### explain_sprintf

const char *explain_sprintf(char *data, const char *format, ...);

The **explain_sprintf** function is used to obtain an explanation of an error returned by the *sprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *sprintf*(3) system call.

*format* The original format, exactly as passed to the *sprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = sprintf(data, format, ...);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_sprintf(data, format, ...));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_sprintf_or_die*(3) function.

### explain_errno_sprintf

const char *explain_errno_sprintf(int errnum, char *data, const char *format, ...);

The **explain_errno_sprintf** function is used to obtain an explanation of an error returned by the *sprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *sprintf*(3) system call.

*format* The original format, exactly as passed to the *sprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = sprintf(data, format, ...);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_sprintf(err, data,
   format, ...));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_sprintf_or_die*(3) function.

**explain_message_sprintf**

void explain_message_sprintf(char *message, int message_size, char *data, const char *format, ...);

The **explain_message_sprintf** function is used to obtain an explanation of an error returned by the *sprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *sprintf*(3) system call.

*format* The original format, exactly as passed to the *sprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = sprintf(data, format, ...);
if (result < 0)
{
    char message[3000];
   explain_message_sprintf(message, sizeof(message), data,
   format, ...);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_sprintf_or_die*(3) function.

**explain_message_errno_sprintf**

void explain_message_errno_sprintf(char *message, int message_size, int errnum, char *data, const char *format, ...);

The **explain_message_errno_sprintf** function is used to obtain an explanation of an error returned by the *sprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *sprintf*(3) system call.

*format* The original format, exactly as passed to the *sprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = sprintf(data, format, ...);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_sprintf(message, sizeof(message), err,
   data, format, ...);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_sprintf_or_die*(3) function.

## SEE ALSO

*sprintf*(3)
formatted output conversion

*explain_sprintf_or_die*(3)
formatted output conversion and report errors

## COPYRIGHT

## NAME

explain_sprintf_or_die − formatted output conversion and report errors

## SYNOPSIS

#include <libexplain/sprintf.h>

int explain_sprintf_or_die(char *data, const char *format, ...);
int explain_sprintf_on_error(char *data, const char *format, ...);

## DESCRIPTION

The **explain_sprintf_or_die** function is used to call the *sprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_sprintf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_sprintf_on_error** function is used to call the *sprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_sprintf*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *sprintf*(3) system call.

*format* The format, exactly as to be passed to the *sprintf*(3) system call.

## RETURN VALUE

The **explain_sprintf_or_die** function only returns on success, see *sprintf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_sprintf_on_error** function always returns the value return by the wrapped *sprintf*(3) system call.

## EXAMPLE

The **explain_sprintf_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_sprintf_or_die(data, format, ...);
```

## SEE ALSO

*sprintf*(3)
formatted output conversion

*explain_sprintf*(3)
explain *sprintf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_stat − explain stat(2) errors

## SYNOPSIS

#include <libexplain/stat.h>
const char *explain_stat(const char *pathname, const struct stat *buf);
void explain_message_stat(char *message, int message_size, const char *pathname, const struct stat *buf);
const char *explain_errno_stat(int errnum, const char *pathname, const struct stat *buf);
void explain_message_errno_stat(char *message, int message_size, int errnum, const char *pathname, const struct stat *buf);

## DESCRIPTION

These functions may be used to obtain explanations for *stat*(2) errors .

### explain_errno_stat

const char *explain_errno_stat(int errnum, const char *pathname, const struct stat *buf);

The explain_errno_stat function is used to obtain an explanation of an error returned by the *stat*(2) function. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (stat(pathname, &buf) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_stat(err, pathname, &buf));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *stat*(2) system call.

*buf* The original buf, exactly as passed to the *stat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_message_errno_stat

void explain_message_errno_stat(char *message, int message_size, int errnum, const char *pathname, const struct stat *buf);

The explain_message_errno_stat function is used to obtain an explanation of an error returned by the *stat*(2) function. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (stat(pathname, &buf) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_stat(message, sizeof(message), err,
        pathname, &buf);
    fprintf(stderr, "%s\n", message);
```

```
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *stat*(2) system call.

*buf* The original buf, exactly as passed to the *stat*(2) system call.

**explain_message_stat**

void explain_message_stat(char *message, int message_size, const char *pathname, const struct stat *buf);

The explain_message_stat function is used to obtain an explanation of an error returned by the *stat*(2) function. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (stat(pathname, &buf) < 0)
{
    char message[3000];
    explain_message_stat(message, sizeof(message), pathname, &buf);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *stat*(2) system call.

*buf* The original buf, exactly as passed to the *stat*(2) system call.

**explain_stat**

const char *explain_stat(const char *pathname, const struct stat * buf);

The explain_stat function is used to obtain an explanation of an error returned by the *stat*(2) function. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (stat(pathname, &buf) < 0)
{
    fprintf(stderr, "%s\n", explain_stat(pathname, &buf));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *stat*(2) system call.

*buf*
: The original buf, exactly as passed to the *stat*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_statfs − explain statfs(2) errors

## SYNOPSIS

#include <libexplain/statfs.h>

const char *explain_statfs(const char *pathname, struct statfs *data);
const char *explain_errno_statfs(int errnum, const char *pathname, struct statfs *data);
void explain_message_statfs(char *message, int message_size, const char *pathname, struct statfs *data);
void explain_message_errno_statfs(char *message, int message_size, int errnum, const char *pathname, struct statfs *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *statfs*(2) system call.

### explain_statfs

const char *explain_statfs(const char *pathname, struct statfs *data);

The **explain_statfs** function is used to obtain an explanation of an error returned by the *statfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *statfs*(2) system call.

*data*
: The original data, exactly as passed to the *statfs*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (statfs(pathname, data) < 0)
{
   fprintf(stderr, "%s\n", explain_statfs(pathname, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_statfs_or_die*(3) function.

### explain_errno_statfs

const char *explain_errno_statfs(int errnum, const char *pathname, struct statfs *data);

The **explain_errno_statfs** function is used to obtain an explanation of an error returned by the *statfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *statfs*(2) system call.

*data*
: The original data, exactly as passed to the *statfs*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (statfs(pathname, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_statfs(err, pathname,
   data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_statfs_or_die*(3) function.

**explain_message_statfs**

void explain_message_statfs(char *message, int message_size, const char *pathname, struct statfs *data);

The **explain_message_statfs** function is used to obtain an explanation of an error returned by the *statfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *statfs*(2) system call.

*data* The original data, exactly as passed to the *statfs*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (statfs(pathname, data) < 0)
{
    char message[3000];
   explain_message_statfs(message, sizeof(message), pathname,
   data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_statfs_or_die*(3) function.

**explain_message_errno_statfs**

void explain_message_errno_statfs(char *message, int message_size, int errnum, const char *pathname, struct statfs *data);

The **explain_message_errno_statfs** function is used to obtain an explanation of an error returned by the *statfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *statfs*(2) system call.

*data* The original data, exactly as passed to the *statfs*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (statfs(pathname, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_statfs(message, sizeof(message), err,
   pathname, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_statfs_or_die*(3) function.

## SEE ALSO

*statfs*(2) get file system statistics

*explain_statfs_or_die*(3)
get file system statistics and report errors

## COPYRIGHT

## NAME

explain_statfs_or_die − get file system statistics and report errors

## SYNOPSIS

#include <libexplain/statfs.h>

void explain_statfs_or_die(const char *pathname, struct statfs *data);
int explain_statfs_on_error(const char *pathname, struct statfs *data);

## DESCRIPTION

The **explain_statfs_or_die** function is used to call the *statfs*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_statfs*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_statfs_on_error** function is used to call the *statfs*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_statfs*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *statfs*(2) system call.

*data*
: The data, exactly as to be passed to the *statfs*(2) system call.

## RETURN VALUE

The **explain_statfs_or_die** function only returns on success, see *statfs*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_statfs_on_error** function always returns the value return by the wrapped *statfs*(2) system call.

## EXAMPLE

The **explain_statfs_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_statfs_or_die(pathname, data);
```

## SEE ALSO

*statfs*(2) get file system statistics

*explain_statfs*(3)
: explain *statfs*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_stat_or_die − get file status and report errors

## SYNOPSIS

#include <libexplain/stat.h>

void explain_stat_or_die(const char *pathname, struct stat *buf);

## DESCRIPTION

The **explain_stat_or_die** function is used to call the *stat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_stat*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_stat_or_die(pathname, buf);
```

*pathname*
: The pathname, exactly as to be passed to the *stat*(2) system call.

*buf*
: The buf, exactly as to be passed to the *stat*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*stat*(2) get file status

*explain_stat*(3)
: explain *stat*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_statvfs − explain *statvfs*(2) errors

## SYNOPSIS

#include <libexplain/statvfs.h>

const char *explain_statvfs(const char *pathname, struct statvfs *data);
const char *explain_errno_statvfs(int errnum, const char *pathname, struct statvfs *data);
void explain_message_statvfs(char *message, int message_size, const char *pathname, struct statvfs *data);
void explain_message_errno_statvfs(char *message, int message_size, int errnum, const char *pathname, struct statvfs *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *statvfs*(2) system call.

### explain_statvfs

const char *explain_statvfs(const char *pathname, struct statvfs *data);

The **explain_statvfs** function is used to obtain an explanation of an error returned by the *statvfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *statvfs*(2) system call.

*data*
: The original data, exactly as passed to the *statvfs*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (statvfs(pathname, data) < 0)
{
   fprintf(stderr, "%s\n", explain_statvfs(pathname, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_statvfs_or_die*(3) function.

### explain_errno_statvfs

const char *explain_errno_statvfs(int errnum, const char *pathname, struct statvfs *data);

The **explain_errno_statvfs** function is used to obtain an explanation of an error returned by the *statvfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *statvfs*(2) system call.

*data*
: The original data, exactly as passed to the *statvfs*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (statvfs(pathname, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_statvfs(err, pathname,
   data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_statvfs_or_die*(3) function.

**explain_message_statvfs**

void explain_message_statvfs(char *message, int message_size, const char *pathname, struct statvfs *data);

The **explain_message_statvfs** function is used to obtain an explanation of an error returned by the *statvfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *statvfs*(2) system call.

*data* The original data, exactly as passed to the *statvfs*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (statvfs(pathname, data) < 0)
{
    char message[3000];
   explain_message_statvfs(message, sizeof(message), pathname,
   data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_statvfs_or_die*(3) function.

**explain_message_errno_statvfs**

void explain_message_errno_statvfs(char *message, int message_size, int errnum, const char *pathname, struct statvfs *data);

The **explain_message_errno_statvfs** function is used to obtain an explanation of an error returned by the *statvfs*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *statvfs*(2) system call.

*data* The original data, exactly as passed to the *statvfs*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (statvfs(pathname, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_statvfs(message, sizeof(message), err,
   pathname, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_statvfs_or_die*(3) function.

## SEE ALSO

*statvfs*(2)
get file system statistics

*explain_statvfs_or_die*(3)
get file system statistics and report errors

## COPYRIGHT

## NAME

explain_statvfs_or_die − get file system statistics and report errors

## SYNOPSIS

#include <libexplain/statvfs.h>

void explain_statvfs_or_die(const char *pathname, struct statvfs *data);
int explain_statvfs_on_error(const char *pathname, struct statvfs *data);

## DESCRIPTION

The **explain_statvfs_or_die** function is used to call the *statvfs*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_statvfs*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_statvfs_on_error** function is used to call the *statvfs*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_statvfs*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *statvfs*(2) system call.

*data*
: The data, exactly as to be passed to the *statvfs*(2) system call.

## RETURN VALUE

The **explain_statvfs_or_die** function only returns on success, see *statvfs*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_statvfs_on_error** function always returns the value return by the wrapped *statvfs*(2) system call.

## EXAMPLE

The **explain_statvfs_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_statvfs_or_die(pathname, data);
```

## SEE ALSO

*statvfs*(2)
: get file system statistics

*explain_statvfs*(3)
: explain *statvfs*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_stime − explain *stime*(2) errors

## SYNOPSIS

#include <libexplain/stime.h>

const char *explain_stime(time_t *t);
const char *explain_errno_stime(int errnum, time_t *t);
void explain_message_stime(char *message, int message_size, time_t *t);
void explain_message_errno_stime(char *message, int message_size, int errnum, time_t *t);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *stime*(2) system call.

### explain_stime

const char *explain_stime(time_t *t);

The **explain_stime** function is used to obtain an explanation of an error returned by the *stime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*t* The original t, exactly as passed to the *stime*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (stime(t) < 0)
{
   fprintf(stderr, "%s\n", explain_stime(t));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_stime_or_die*(3) function.

### explain_errno_stime

const char *explain_errno_stime(int errnum, time_t *t);

The **explain_errno_stime** function is used to obtain an explanation of an error returned by the *stime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*t* The original t, exactly as passed to the *stime*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (stime(t) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_stime(err, t));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_stime_or_die*(3) function.

**explain_message_stime**

void explain_message_stime(char *message, int message_size, time_t *t);

The **explain_message_stime** function is used to obtain an explanation of an error returned by the *stime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*t* The original t, exactly as passed to the *stime*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (stime(t) < 0)
{
    char message[3000];
    explain_message_stime(message, sizeof(message), t);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_stime_or_die*(3) function.

**explain_message_errno_stime**

void explain_message_errno_stime(char *message, int message_size, int errnum, time_t *t);

The **explain_message_errno_stime** function is used to obtain an explanation of an error returned by the *stime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*t* The original t, exactly as passed to the *stime*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (stime(t) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_stime(message, sizeof(message), err, t);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_stime_or_die*(3) function.

## SEE ALSO

*stime*(2) set system time

*explain_stime_or_die*(3)
set system time and report errors

## COPYRIGHT

## NAME

explain_stime_or_die − set system time and report errors

## SYNOPSIS

#include <libexplain/stime.h>

void explain_stime_or_die(time_t *t);
int explain_stime_on_error(time_t *t);

## DESCRIPTION

The **explain_stime_or_die** function is used to call the *stime*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_stime*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_stime_on_error** function is used to call the *stime*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_stime*(3) function, but still returns to the caller.

*t* The t, exactly as to be passed to the *stime*(2) system call.

## RETURN VALUE

The **explain_stime_or_die** function only returns on success, see *stime*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_stime_on_error** function always returns the value return by the wrapped *stime*(2) system call.

## EXAMPLE

The **explain_stime_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_stime_or_die(t);
```

## SEE ALSO

*stime*(2) set system time

*explain_stime*(3)
explain *stime*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strdup − explain strdup(3) errors

## SYNOPSIS

#include <libexplain/strdup.h>

const char *explain_strdup(const char *data);
const char *explain_errno_strdup(int errnum, const char *data);
void explain_message_strdup(char *message, int message_size, const char *data);
void explain_message_errno_strdup(char *message, int message_size, int errnum, const char *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strdup*(3) system call.

### explain_strdup

const char *explain_strdup(const char *data);

The **explain_strdup** function is used to obtain an explanation of an error returned by the *strdup*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *strdup*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = strdup(data);
if (!result)
{
   fprintf(stderr, "%s\n", explain_strdup(data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strdup_or_die*(3) function.

### explain_errno_strdup

const char *explain_errno_strdup(int errnum, const char *data);

The **explain_errno_strdup** function is used to obtain an explanation of an error returned by the *strdup*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *strdup*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = strdup(data);
```

```
if (!result)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_strdup(err, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strdup_or_die*(3) function.

**explain_message_strdup**

void explain_message_strdup(char *message, int message_size, const char *data);

The **explain_message_strdup** function is used to obtain an explanation of an error returned by the *strdup*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *strdup*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = strdup(data);
if (!result)
{
    char message[3000];
    explain_message_strdup(message, sizeof(message), data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strdup_or_die*(3) function.

**explain_message_errno_strdup**

void explain_message_errno_strdup(char *message, int message_size, int errnum, const char *data);

The **explain_message_errno_strdup** function is used to obtain an explanation of an error returned by the *strdup*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *strdup*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = strdup(data);
if (!result)
{
    int err = errno;
    char message[3000];
    explain_message_errno_strdup(message, sizeof(message), err,
```

```
                data);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_strdup_or_die*(3) function.

## SEE ALSO

*strdup*(3)
duplicate a string

*explain_strdup_or_die*(3)
duplicate a string and report errors

## COPYRIGHT

## NAME

explain_strdup_or_die − duplicate a string and report errors

## SYNOPSIS

#include <libexplain/strdup.h>

char *explain_strdup_or_die(const char *data);
char *explain_strdup_on_error(const char *data);

## DESCRIPTION

The **explain_strdup_or_die** function is used to call the *strdup*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strdup*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strdup_on_error** function is used to call the *strdup*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strdup*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *strdup*(3) system call.

## RETURN VALUE

The **explain_strdup_or_die** function only returns on success, see *strdup*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strdup_on_error** function always returns the value return by the wrapped *strdup*(3) system call.

## EXAMPLE

The **explain_strdup_or_die** function is intended to be used in a fashion similar to the following example:

```
char *result = explain_strdup_or_die(data);
```

## SEE ALSO

*strdup*(3)
duplicate a string

*explain_strdup*(3)
explain *strdup*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strndup − explain strndup(3) errors

## SYNOPSIS

#include <libexplain/strndup.h>

const char *explain_strndup(const char *data, size_t data_size);
const char *explain_errno_strndup(int errnum, const char *data, size_t data_size);
void explain_message_strndup(char *message, int message_size, const char *data, size_t data_size);
void explain_message_errno_strndup(char *message, int message_size, int errnum, const char *data, size_t data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strndup*(3) system call.

### explain_strndup

const char *explain_strndup(const char *data, size_t data_size);

The **explain_strndup** function is used to obtain an explanation of an error returned by the *strndup*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *strndup*(3) system call.

*data_size*
The original data_size, exactly as passed to the *strndup*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = strndup(data, data_size);
if (!result)
{
   fprintf(stderr, "%s\n", explain_strndup(data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strndup_or_die*(3) function.

### explain_errno_strndup

const char *explain_errno_strndup(int errnum, const char *data, size_t data_size);

The **explain_errno_strndup** function is used to obtain an explanation of an error returned by the *strndup*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *strndup*(3) system call.

*data_size*
The original data_size, exactly as passed to the *strndup*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = strndup(data, data_size);
if (!result)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_strndup(err, data,
   data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strndup_or_die*(3) function.

**explain_message_strndup**

void explain_message_strndup(char *message, int message_size, const char *data, size_t data_size);

The **explain_message_strndup** function is used to obtain an explanation of an error returned by the *strndup*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *strndup*(3) system call.

*data_size*
The original data_size, exactly as passed to the *strndup*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = strndup(data, data_size);
if (!result)
{
    char message[3000];
   explain_message_strndup(message, sizeof(message), data,
   data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strndup_or_die*(3) function.

**explain_message_errno_strndup**

void explain_message_errno_strndup(char *message, int message_size, int errnum, const char *data, size_t data_size);

The **explain_message_errno_strndup** function is used to obtain an explanation of an error returned by the *strndup*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *strndup*(3) system call.

*data_size*
The original data_size, exactly as passed to the *strndup*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = strndup(data, data_size);
if (!result)
{
    int err = errno;
    char message[3000];
   explain_message_errno_strndup(message, sizeof(message), err,
   data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strndup_or_die*(3) function.

## SEE ALSO

*strndup*(3)
duplicate a string

*explain_strndup_or_die*(3)
duplicate a string and report errors

## COPYRIGHT

## NAME

explain_strndup_or_die − duplicate a string and report errors

## SYNOPSIS

#include <libexplain/strndup.h>

char *explain_strndup_or_die(const char *data, size_t data_size);
char *explain_strndup_on_error(const char *data, size_t data_size);

## DESCRIPTION

The **explain_strndup_or_die** function is used to call the *strndup*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strndup*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strndup_on_error** function is used to call the *strndup*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strndup*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *strndup*(3) system call.

*data_size*
The data_size, exactly as to be passed to the *strndup*(3) system call.

## RETURN VALUE

The **explain_strndup_or_die** function only returns on success, see *strndup*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strndup_on_error** function always returns the value return by the wrapped *strndup*(3) system call.

## EXAMPLE

The **explain_strndup_or_die** function is intended to be used in a fashion similar to the following example:

```
char *result = explain_strndup_or_die(data, data_size);
```

## SEE ALSO

*strndup*(3)
duplicate a string

*explain_strndup*(3)
explain *strndup*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strtod − explain strtod(3) errors

## SYNOPSIS

#include <libexplain/strtod.h>

const char *explain_strtod(const char *nptr, char **endptr);
const char *explain_errno_strtod(int errnum, const char *nptr, char **endptr);
void explain_message_strtod(char *message, int message_size, const char *nptr, char **endptr);
void explain_message_errno_strtod(char *message, int message_size, int errnum, const char *nptr, char **endptr);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strtod*(3) system call.

### explain_strtod

const char *explain_strtod(const char *nptr, char **endptr);

The **explain_strtod** function is used to obtain an explanation of an error returned by the *strtod*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*nptr* The original nptr, exactly as passed to the *strtod*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtod*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
double result = strtod(nptr, endptr);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_strtod(nptr, endptr));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtod_or_die*(3) function.

### explain_errno_strtod

const char *explain_errno_strtod(int errnum, const char *nptr, char **endptr);

The **explain_errno_strtod** function is used to obtain an explanation of an error returned by the *strtod*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtod*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtod*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
double result = strtod(nptr, endptr);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_strtod(err, nptr,
   endptr));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtod_or_die*(3) function.

**explain_message_strtod**

void explain_message_strtod(char *message, int message_size, const char *nptr, char **endptr);

The **explain_message_strtod** function is used to obtain an explanation of an error returned by the *strtod*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nptr* The original nptr, exactly as passed to the *strtod*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtod*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
double result = strtod(nptr, endptr);
if (result < 0)
{
    char message[3000];
   explain_message_strtod(message, sizeof(message), nptr,
   endptr);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtod_or_die*(3) function.

**explain_message_errno_strtod**

void explain_message_errno_strtod(char *message, int message_size, int errnum, const char *nptr, char **endptr);

The **explain_message_errno_strtod** function is used to obtain an explanation of an error returned by the *strtod*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtod*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtod*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
double result = strtod(nptr, endptr);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_strtod(message, sizeof(message), err,
   nptr, endptr);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtod_or_die*(3) function.

## SEE ALSO

*strtod*(3) convert ASCII string to floating-point number

*explain_strtod_or_die*(3)
convert ASCII string to floating-point number and report errors

## COPYRIGHT

## NAME

explain_strtod_or_die − convert string to number and report errors

## SYNOPSIS

#include <libexplain/strtod.h>

double explain_strtod_or_die(const char *nptr, char **endptr);
double explain_strtod_on_error(const char *nptr, char **endptr))

## DESCRIPTION

The **explain_strtod_or_die** function is used to call the *strtod*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtod*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strtod_on_error** function is used to call the *strtod*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtod*(3) function, but still returns to the caller.

*nptr* The nptr, exactly as to be passed to the *strtod*(3) system call.

*endptr* The endptr, exactly as to be passed to the *strtod*(3) system call.

## RETURN VALUE

The **explain_strtod_or_die** function only returns on success, see *strtod*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strtod_on_error** function always returns the value return by the wrapped *strtod*(3) system call.

## EXAMPLE

The **explain_strtod_or_die** function is intended to be used in a fashion similar to the following example:

```
double result = explain_strtod_or_die(nptr, endptr);
```

## SEE ALSO

*strtod*(3) convert ASCII string to floating-point number

*explain_strtod*(3)
explain *strtod*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strtof − explain strtof(3) errors

## SYNOPSIS

#include <libexplain/strtof.h>

const char *explain_strtof(const char *nptr, char **endptr);
const char *explain_errno_strtof(int errnum, const char *nptr, char **endptr);
void explain_message_strtof(char *message, int message_size, const char *nptr, char **endptr);
void explain_message_errno_strtof(char *message, int message_size, int errnum, const char *nptr, char **endptr);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strtof*(3) system call.

### explain_strtof

const char *explain_strtof(const char *nptr, char **endptr);

The **explain_strtof** function is used to obtain an explanation of an error returned by the *strtof*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*nptr* The original nptr, exactly as passed to the *strtof*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtof*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
float result = strtof(nptr, endptr);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_strtof(nptr, endptr));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtof_or_die*(3) function.

### explain_errno_strtof

const char *explain_errno_strtof(int errnum, const char *nptr, char **endptr);

The **explain_errno_strtof** function is used to obtain an explanation of an error returned by the *strtof*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtof*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtof*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
float result = strtof(nptr, endptr);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_strtof(err, nptr,
   endptr));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtof_or_die*(3) function.

**explain_message_strtof**

void explain_message_strtof(char *message, int message_size, const char *nptr, char **endptr);

The **explain_message_strtof** function is used to obtain an explanation of an error returned by the *strtof*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nptr* The original nptr, exactly as passed to the *strtof*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtof*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
float result = strtof(nptr, endptr);
if (result < 0)
{
    char message[3000];
   explain_message_strtof(message, sizeof(message), nptr,
   endptr);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtof_or_die*(3) function.

**explain_message_errno_strtof**

void explain_message_errno_strtof(char *message, int message_size, int errnum, const char *nptr, char **endptr);

The **explain_message_errno_strtof** function is used to obtain an explanation of an error returned by the *strtof*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtof*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtof*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
float result = strtof(nptr, endptr);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_strtof(message, sizeof(message), err,
   nptr, endptr);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtof_or_die*(3) function.

## SEE ALSO

*strtof*(3) convert ASCII string to floating-point number

*explain_strtof_or_die*(3)
convert ASCII string to floating-point number and report errors

## COPYRIGHT

## NAME

explain_strtof_or_die − convert string to number and report errors

## SYNOPSIS

#include <libexplain/strtof.h>

float explain_strtof_or_die(const char *nptr, char **endptr);
float explain_strtof_on_error(const char *nptr, char **endptr))

## DESCRIPTION

The **explain_strtof_or_die** function is used to call the *strtof*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtof*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strtof_on_error** function is used to call the *strtof*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtof*(3) function, but still returns to the caller.

*nptr* The nptr, exactly as to be passed to the *strtof*(3) system call.

*endptr* The endptr, exactly as to be passed to the *strtof*(3) system call.

## RETURN VALUE

The **explain_strtof_or_die** function only returns on success, see *strtof*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strtof_on_error** function always returns the value return by the wrapped *strtof*(3) system call.

## EXAMPLE

The **explain_strtof_or_die** function is intended to be used in a fashion similar to the following example:

```
float result = explain_strtof_or_die(nptr, endptr);
```

## SEE ALSO

*strtof*(3) convert ASCII string to floating-point number

*explain_strtof*(3)
explain *strtof*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strtol − explain strtol(3) errors

## SYNOPSIS

#include <libexplain/strtol.h>

const char *explain_strtol(const char *nptr, char **endptr, int base);
const char *explain_errno_strtol(int errnum, const char *nptr, char **endptr, int base);
void explain_message_strtol(char *message, int message_size, const char *nptr, char **endptr, int base);
void explain_message_errno_strtol(char *message, int message_size, int errnum, const char *nptr, char **endptr, int base);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strtol*(3) system call.

### explain_strtol

const char *explain_strtol(const char *nptr, char **endptr, int base);

The **explain_strtol** function is used to obtain an explanation of an error returned by the *strtol*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*nptr* The original nptr, exactly as passed to the *strtol*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtol*(3) system call.

*base* The original base, exactly as passed to the *strtol*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = strtol(nptr, endptr, base);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_strtol(nptr, endptr, base));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtol_or_die*(3) function.

### explain_errno_strtol

const char *explain_errno_strtol(int errnum, const char *nptr, char **endptr, int base);

The **explain_errno_strtol** function is used to obtain an explanation of an error returned by the *strtol*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtol*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtol*(3) system call.

*base* The original base, exactly as passed to the *strtol*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = strtol(nptr, endptr, base);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_strtol(err, nptr,
   endptr, base));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtol_or_die*(3) function.

**explain_message_strtol**

void explain_message_strtol(char *message, int message_size, const char *nptr, char **endptr, int base);

The **explain_message_strtol** function is used to obtain an explanation of an error returned by the *strtol*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nptr* The original nptr, exactly as passed to the *strtol*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtol*(3) system call.

*base* The original base, exactly as passed to the *strtol*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = strtol(nptr, endptr, base);
if (result < 0)
{
    char message[3000];
   explain_message_strtol(message, sizeof(message), nptr, endptr,
   base);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtol_or_die*(3) function.

**explain_message_errno_strtol**

void explain_message_errno_strtol(char *message, int message_size, int errnum, const char *nptr, char **endptr, int base);

The **explain_message_errno_strtol** function is used to obtain an explanation of an error returned by the *strtol*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtol*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtol*(3) system call.

*base* The original base, exactly as passed to the *strtol*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long result = strtol(nptr, endptr, base);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_strtol(message, sizeof(message), err,
   nptr, endptr, base);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtol_or_die*(3) function.

## SEE ALSO

*strtol*(3) convert a string to a long integer

*explain_strtol_or_die*(3)
convert a string to a long integer and report errors

## COPYRIGHT

## NAME

explain_strtold − explain strtold(3) errors

## SYNOPSIS

#include <libexplain/strtold.h>

const char *explain_strtold(const char *nptr, char **endptr);
const char *explain_errno_strtold(int errnum, const char *nptr, char **endptr);
void explain_message_strtold(char *message, int message_size, const char *nptr, char **endptr);
void explain_message_errno_strtold(char *message, int message_size, int errnum, const char *nptr, char **endptr);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strtold*(3) system call.

### explain_strtold

const char *explain_strtold(const char *nptr, char **endptr);

The **explain_strtold** function is used to obtain an explanation of an error returned by the *strtold*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*nptr* The original nptr, exactly as passed to the *strtold*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtold*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long double result = strtold(nptr, endptr);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_strtold(nptr, endptr));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtold_or_die*(3) function.

### explain_errno_strtold

const char *explain_errno_strtold(int errnum, const char *nptr, char **endptr);

The **explain_errno_strtold** function is used to obtain an explanation of an error returned by the *strtold*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtold*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtold*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long double result = strtold(nptr, endptr);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_strtold(err, nptr,
   endptr));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtold_or_die*(3) function.

**explain_message_strtold**

void explain_message_strtold(char *message, int message_size, const char *nptr, char **endptr);

The **explain_message_strtold** function is used to obtain an explanation of an error returned by the *strtold*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nptr* The original nptr, exactly as passed to the *strtold*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtold*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long double result = strtold(nptr, endptr);
if (result < 0)
{
    char message[3000];
   explain_message_strtold(message, sizeof(message), nptr,
   endptr);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtold_or_die*(3) function.

**explain_message_errno_strtold**

void explain_message_errno_strtold(char *message, int message_size, int errnum, const char *nptr, char **endptr);

The **explain_message_errno_strtold** function is used to obtain an explanation of an error returned by the *strtold*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtold*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtold*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long double result = strtold(nptr, endptr);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_strtold(message, sizeof(message), err,
   nptr, endptr);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtold_or_die*(3) function.

## SEE ALSO

*strtold*(3)
convert ASCII string to floating-point number

*explain_strtold_or_die*(3)
convert ASCII string to floating-point number and report errors

## COPYRIGHT

## NAME

explain_strtold_or_die − convert string to number and report errors

## SYNOPSIS

#include <libexplain/strtold.h>

long double explain_strtold_or_die(const char *nptr, char **endptr);
long double explain_strtold_on_error(const char *nptr, char **endptr))

## DESCRIPTION

The **explain_strtold_or_die** function is used to call the *strtold*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtold*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strtold_on_error** function is used to call the *strtold*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtold*(3) function, but still returns to the caller.

*nptr* The nptr, exactly as to be passed to the *strtold*(3) system call.

*endptr* The endptr, exactly as to be passed to the *strtold*(3) system call.

## RETURN VALUE

The **explain_strtold_or_die** function only returns on success, see *strtold*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strtold_on_error** function always returns the value return by the wrapped *strtold*(3) system call.

## EXAMPLE

The **explain_strtold_or_die** function is intended to be used in a fashion similar to the following example:

```
long double result = explain_strtold_or_die(nptr, endptr);
```

## SEE ALSO

*strtold*(3)
convert ASCII string to floating-point number

*explain_strtold*(3)
explain *strtold*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strtoll − explain strtoll(3) errors

## SYNOPSIS

#include <libexplain/strtoll.h>

const char *explain_strtoll(const char *nptr, char **endptr, int base);
const char *explain_errno_strtoll(int errnum, const char *nptr, char **endptr, int base);
void explain_message_strtoll(char *message, int message_size, const char *nptr, char **endptr, int base);
void explain_message_errno_strtoll(char *message, int message_size, int errnum, const char *nptr, char **endptr, int base);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strtoll*(3) system call.

### explain_strtoll

const char *explain_strtoll(const char *nptr, char **endptr, int base);

The **explain_strtoll** function is used to obtain an explanation of an error returned by the *strtoll*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*nptr* The original nptr, exactly as passed to the *strtoll*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoll*(3) system call.

*base* The original base, exactly as passed to the *strtoll*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long long result = strtoll(nptr, endptr, base);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_strtoll(nptr, endptr, base));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoll_or_die*(3) function.

### explain_errno_strtoll

const char *explain_errno_strtoll(int errnum, const char *nptr, char **endptr, int base);

The **explain_errno_strtoll** function is used to obtain an explanation of an error returned by the *strtoll*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtoll*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoll*(3) system call.

*base* The original base, exactly as passed to the *strtoll*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long long result = strtoll(nptr, endptr, base);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_strtoll(err, nptr,
   endptr, base));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoll_or_die*(3) function.

**explain_message_strtoll**

void explain_message_strtoll(char *message, int message_size, const char *nptr, char **endptr, int base);

The **explain_message_strtoll** function is used to obtain an explanation of an error returned by the *strtoll*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nptr* The original nptr, exactly as passed to the *strtoll*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoll*(3) system call.

*base* The original base, exactly as passed to the *strtoll*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long long result = strtoll(nptr, endptr, base);
if (result < 0)
{
    char message[3000];
   explain_message_strtoll(message, sizeof(message), nptr,
   endptr, base);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoll_or_die*(3) function.

**explain_message_errno_strtoll**

void explain_message_errno_strtoll(char *message, int message_size, int errnum, const char *nptr, char **endptr, int base);

The **explain_message_errno_strtoll** function is used to obtain an explanation of an error returned by the *strtoll*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtoll*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoll*(3) system call.

*base* The original base, exactly as passed to the *strtoll*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
long long result = strtoll(nptr, endptr, base);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_strtoll(message, sizeof(message), err,
   nptr, endptr, base);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoll_or_die*(3) function.

## SEE ALSO

*strtoll*(3)
convert a string to a long integer

*explain_strtoll_or_die*(3)
convert a string to a long integer and report errors

## COPYRIGHT

## NAME

explain_strtoll_or_die − convert a string to a long integer and report errors

## SYNOPSIS

#include <libexplain/strtoll.h>

long long explain_strtoll_or_die(const char *nptr, char **endptr, int base);
long long explain_strtoll_on_error(const char *nptr, char **endptr, int base))

## DESCRIPTION

The **explain_strtoll_or_die** function is used to call the *strtoll*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtoll*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strtoll_on_error** function is used to call the *strtoll*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtoll*(3) function, but still returns to the caller.

*nptr* The nptr, exactly as to be passed to the *strtoll*(3) system call.

*endptr* The endptr, exactly as to be passed to the *strtoll*(3) system call.

*base* The base, exactly as to be passed to the *strtoll*(3) system call.

## RETURN VALUE

The **explain_strtoll_or_die** function only returns on success, see *strtoll*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strtoll_on_error** function always returns the value return by the wrapped *strtoll*(3) system call.

## EXAMPLE

The **explain_strtoll_or_die** function is intended to be used in a fashion similar to the following example:

```
long long result = explain_strtoll_or_die(nptr, endptr, base);
```

## SEE ALSO

*strtoll*(3)
convert a string to a long integer

*explain_strtoll*(3)
explain *strtoll*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strtol_or_die − convert a string to a long integer and report errors

## SYNOPSIS

#include <libexplain/strtol.h>

long explain_strtol_or_die(const char *nptr, char **endptr, int base);
long explain_strtol_on_error(const char *nptr, char **endptr, int base))

## DESCRIPTION

The **explain_strtol_or_die** function is used to call the *strtol*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtol*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strtol_on_error** function is used to call the *strtol*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtol*(3) function, but still returns to the caller.

*nptr* The nptr, exactly as to be passed to the *strtol*(3) system call.

*endptr* The endptr, exactly as to be passed to the *strtol*(3) system call.

*base* The base, exactly as to be passed to the *strtol*(3) system call.

## RETURN VALUE

The **explain_strtol_or_die** function only returns on success, see *strtol*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strtol_on_error** function always returns the value return by the wrapped *strtol*(3) system call.

## EXAMPLE

The **explain_strtol_or_die** function is intended to be used in a fashion similar to the following example:

```
long result = explain_strtol_or_die(nptr, endptr, base);
```

## SEE ALSO

*strtol*(3) convert a string to a long integer

*explain_strtol*(3)
explain *strtol*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strtoul − explain strtoul(3) errors

## SYNOPSIS

#include <libexplain/strtoul.h>

const char *explain_strtoul(const char *nptr, char **endptr, int base);
const char *explain_errno_strtoul(int errnum, const char *nptr, char **endptr, int base);
void explain_message_strtoul(char *message, int message_size, const char *nptr, char **endptr, int base);
void explain_message_errno_strtoul(char *message, int message_size, int errnum, const char *nptr, char **endptr, int base);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strtoul*(3) system call.

### explain_strtoul

const char *explain_strtoul(const char *nptr, char **endptr, int base);

The **explain_strtoul** function is used to obtain an explanation of an error returned by the *strtoul*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*nptr* The original nptr, exactly as passed to the *strtoul*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoul*(3) system call.

*base* The original base, exactly as passed to the *strtoul*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
unsigned long result = strtoul(nptr, endptr, base);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_strtoul(nptr, endptr, base));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoul_or_die*(3) function.

### explain_errno_strtoul

const char *explain_errno_strtoul(int errnum, const char *nptr, char **endptr, int base);

The **explain_errno_strtoul** function is used to obtain an explanation of an error returned by the *strtoul*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtoul*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoul*(3) system call.

*base* The original base, exactly as passed to the *strtoul*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
unsigned long result = strtoul(nptr, endptr, base);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_strtoul(err, nptr,
   endptr, base));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoul_or_die*(3) function.

**explain_message_strtoul**

void explain_message_strtoul(char *message, int message_size, const char *nptr, char **endptr, int base);

The **explain_message_strtoul** function is used to obtain an explanation of an error returned by the *strtoul*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nptr* The original nptr, exactly as passed to the *strtoul*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoul*(3) system call.

*base* The original base, exactly as passed to the *strtoul*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
unsigned long result = strtoul(nptr, endptr, base);
if (result < 0)
{
    char message[3000];
   explain_message_strtoul(message, sizeof(message), nptr,
   endptr, base);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoul_or_die*(3) function.

**explain_message_errno_strtoul**

void explain_message_errno_strtoul(char *message, int message_size, int errnum, const char *nptr, char **endptr, int base);

The **explain_message_errno_strtoul** function is used to obtain an explanation of an error returned by the *strtoul*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtoul*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoul*(3) system call.

*base* The original base, exactly as passed to the *strtoul*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
unsigned long result = strtoul(nptr, endptr, base);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_strtoul(message, sizeof(message), err,
   nptr, endptr, base);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoul_or_die*(3) function.

## SEE ALSO

*strtoul*(3)
convert a string to an unsigned long integer

*explain_strtoul_or_die*(3)
convert a string to an unsigned long integer and report errors

## COPYRIGHT

## NAME

explain_strtoull − explain strtoull(3) errors

## SYNOPSIS

#include <libexplain/strtoull.h>

const char *explain_strtoull(const char *nptr, char **endptr, int base);
const char *explain_errno_strtoull(int errnum, const char *nptr, char **endptr, int base);
void explain_message_strtoull(char *message, int message_size, const char *nptr, char **endptr, int base);
void explain_message_errno_strtoull(char *message, int message_size, int errnum, const char *nptr, char **endptr, int base);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *strtoull*(3) system call.

### explain_strtoull

const char *explain_strtoull(const char *nptr, char **endptr, int base);

The **explain_strtoull** function is used to obtain an explanation of an error returned by the *strtoull*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*nptr* The original nptr, exactly as passed to the *strtoull*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoull*(3) system call.

*base* The original base, exactly as passed to the *strtoull*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
unsigned long long result = strtoull(nptr, endptr, base);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_strtoull(nptr, endptr, base));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoull_or_die*(3) function.

### explain_errno_strtoull

const char *explain_errno_strtoull(int errnum, const char *nptr, char **endptr, int base);

The **explain_errno_strtoull** function is used to obtain an explanation of an error returned by the *strtoull*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtoull*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoull*(3) system call.

*base* The original base, exactly as passed to the *strtoull*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
unsigned long long result = strtoull(nptr, endptr, base);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_strtoull(err, nptr,
   endptr, base));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoull_or_die*(3) function.

**explain_message_strtoull**

void explain_message_strtoull(char *message, int message_size, const char *nptr, char **endptr, int base);

The **explain_message_strtoull** function is used to obtain an explanation of an error returned by the *strtoull*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*nptr* The original nptr, exactly as passed to the *strtoull*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoull*(3) system call.

*base* The original base, exactly as passed to the *strtoull*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
unsigned long long result = strtoull(nptr, endptr, base);
if (result < 0)
{
    char message[3000];
   explain_message_strtoull(message, sizeof(message), nptr,
   endptr, base);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoull_or_die*(3) function.

**explain_message_errno_strtoull**

void explain_message_errno_strtoull(char *message, int message_size, int errnum, const char *nptr, char **endptr, int base);

The **explain_message_errno_strtoull** function is used to obtain an explanation of an error returned by the *strtoull*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*nptr* The original nptr, exactly as passed to the *strtoull*(3) system call.

*endptr* The original endptr, exactly as passed to the *strtoull*(3) system call.

*base* The original base, exactly as passed to the *strtoull*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
unsigned long long result = strtoull(nptr, endptr, base);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_strtoull(message, sizeof(message), err,
   nptr, endptr, base);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_strtoull_or_die*(3) function.

## SEE ALSO

*strtoull*(3)
convert a string to an unsigned long integer

*explain_strtoull_or_die*(3)
convert a string to an unsigned long integer and report errors

## COPYRIGHT

## NAME

explain_strtoull_or_die − convert string to integer and report errors

## SYNOPSIS

#include <libexplain/strtoull.h>

unsigned long long explain_strtoull_or_die(const char *nptr, char **endptr, int base);
unsigned long long explain_strtoull_on_error(const char *nptr, char **endptr, int base))

## DESCRIPTION

The **explain_strtoull_or_die** function is used to call the *strtoull*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtoull*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strtoull_on_error** function is used to call the *strtoull*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtoull*(3) function, but still returns to the caller.

*nptr* The nptr, exactly as to be passed to the *strtoull*(3) system call.

*endptr* The endptr, exactly as to be passed to the *strtoull*(3) system call.

*base* The base, exactly as to be passed to the *strtoull*(3) system call.

## RETURN VALUE

The **explain_strtoull_or_die** function only returns on success, see *strtoull*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strtoull_on_error** function always returns the value return by the wrapped *strtoull*(3) system call.

## EXAMPLE

The **explain_strtoull_or_die** function is intended to be used in a fashion similar to the following example:

```
unsigned long long result = explain_strtoull_or_die(nptr, endptr, base);
```

## SEE ALSO

*strtoull*(3)
convert a string to an unsigned long integer

*explain_strtoull*(3)
explain *strtoull*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_strtoul_or_die − convert string to unsigned long and report errors

## SYNOPSIS

#include <libexplain/strtoul.h>

unsigned long explain_strtoul_or_die(const char *nptr, char **endptr, int base);
unsigned long explain_strtoul_on_error(const char *nptr, char **endptr, int base))

## DESCRIPTION

The **explain_strtoul_or_die** function is used to call the *strtoul*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtoul*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_strtoul_on_error** function is used to call the *strtoul*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_strtoul*(3) function, but still returns to the caller.

*nptr* The nptr, exactly as to be passed to the *strtoul*(3) system call.

*endptr* The endptr, exactly as to be passed to the *strtoul*(3) system call.

*base* The base, exactly as to be passed to the *strtoul*(3) system call.

## RETURN VALUE

The **explain_strtoul_or_die** function only returns on success, see *strtoul*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_strtoul_on_error** function always returns the value return by the wrapped *strtoul*(3) system call.

## EXAMPLE

The **explain_strtoul_or_die** function is intended to be used in a fashion similar to the following example:

```
unsigned long result = explain_strtoul_or_die(nptr, endptr, base);
```

## SEE ALSO

*strtoul*(3)
convert a string to an unsigned long integer

*explain_strtoul*(3)
explain *strtoul*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_symlink − explain symlink(2) errors

## SYNOPSIS

#include <libexplain/symlink.h>
const char *explain_symlink(const char *oldpath, const char *newpath);
const char *explain_errno_symlink(int errnum, const char *oldpath, const char *newpath);
void explain_message_symlink(char *message, int message_size, const char *oldpath, const char *newpath);
void explain_message_errno_symlink(char *message, int message_size, int errnum, const char *oldpath, const char *newpath);

## DESCRIPTION

These functions may be used to obtain explanations for *symlink*(2) errors.

### explain_symlink

const char *explain_symlink(const char *oldpath, const char *newpath);

The explain_symlink function is used to obtain an explanation of an error returned by the *symlink*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (symlink(oldpath, rewpath) < 0)
{
    fprintf(stderr, '%s0, explain_symlink(oldpath, newpath));
    exit(EXIT_FAILURE);
}
```

*oldpath* The original oldpath, exactly as passed to the *symlink*(2) system call.

*newpath* The original newpath, exactly as passed to the *symlink*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_symlink

const char *explain_errno_symlink(int errnum, const char *oldpath, const char * newpath);

The explain_errno_symlink function is used to obtain an explanation of an error returned by the *symlink*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (symlink(oldpath, newpath) < 0)
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_symlink(err, oldpath,
        newpath));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*oldpath* The original oldpath, exactly as passed to the *symlink*(2) system call.

*newpath* The original newpath, exactly as passed to the *symlink*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_symlink**

void explain_message_symlink(char *message, int message_size, const char *oldpath, const char *newpath);

The explain_message_symlink function is used to obtain an explanation of an error returned by the *symlink*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno]fP global variable will be used to obtain the error value to be decoded.*

This function is intended to be used in a fashion similar to the following example:

```
if (symlink(oldpath, newpath) < 0)
{
    char message[3000];
    explain_message_symlink(message, sizeof(message), oldpath,
        newpath);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*oldpath* The original oldpath, exactly as passed to the *symlink*(2) system call.

*newpath* The original newpath, exactly as passed to the *symlink*(2) system call.

**explain_message_errno_symlink**

void explain_message_errno_symlink(char *message, int message_size, int errnum, const char *oldpath, const char *newpath);

The explain_message_errno_symlink function is used to obtain an explanation of an error returned by the *symlink*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (symlink(oldpath, newpath) < 0)
{
  int err = errno;
  char message[3000];
  explain_message_errno_symlink(message, sizeof(message), err,
    oldpath, newpath);
  fprintf(stderr, '%s0, message);
  exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*oldpath* The original oldpath, exactly as passed to the *symlink*(2) system call.

*newpath* The original newpath, exactly as passed to the *symlink*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_symlink_or_die − make a new name for a file and report errors

## SYNOPSIS

#include <libexplain/symlink.h>

void explain_symlink_or_die(const char *oldpath, const char *newpath);

## DESCRIPTION

The **explain_symlink_or_die** function is used to call the *symlink*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_symlink*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_symlink_or_die(oldpath, newpath);
```

*oldpath* The oldpath, exactly as to be passed to the *symlink*(2) system call.

*newpath* The newpath, exactly as to be passed to the *symlink*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*symlink*(2)
make a new name for a file

*explain_symlink*(3)
explain *symlink*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_system − explain system(3) errors

## SYNOPSIS

#include <libexplain/system.h>

const char *explain_system(const char *command);
const char *explain_errno_system(int errnum, const char *command);
void explain_message_system(char *message, int message_size, const char *command);
void explain_message_errno_system(char *message, int message_size, int errnum, const char *command);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *system*(3) system call.

### explain_system

const char *explain_system(const char *command);

The **explain_system** function is used to obtain an explanation of an error returned by the *system*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (system(command) < 0)
{
    fprintf(stderr, "%s\n", explain_system(command));
    exit(EXIT_FAILURE);
}
```

*command*
: The original command, exactly as passed to the *system*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_system

const char *explain_errno_system(int errnum, const char *command);

The **explain_errno_system** function is used to obtain an explanation of an error returned by the *system*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (system(command) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_system(err, command));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*command*
: The original command, exactly as passed to the *system*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_system**

void explain_message_system(char *message, int message_size, const char *command);

The **explain_message_system** function may be used to obtain an explanation of an error returned by the *system*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (system(command) < 0)
{
    char message[3000];
    explain_message_system(message, sizeof(message), command);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*command*
The original command, exactly as passed to the *system*(3) system call.

**explain_message_errno_system**

void explain_message_errno_system(char *message, int message_size, int errnum, const char *command);

The **explain_message_errno_system** function may be used to obtain an explanation of an error returned by the *system*(3) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (system(command) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_system(message, sizeof(message), err, command);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*command*
The original command, exactly as passed to the *system*(3) system call.

## SEE ALSO

*system*(3)
execute a shell command

*explain_system_or_die*(3)
execute a shell command and report errors

## COPYRIGHT

## NAME

explain_system_or_die − execute a shell command and report errors

## SYNOPSIS

#include <libexplain/system.h>

void explain_system_or_die(const char *command);
void explain_system_success_or_die(const char *command);
int explain_system_success(const char *command);

## DESCRIPTION

These functions may be used to execute commands via the *system*(3) function, and report the results.

**explain_system_or_die**

void explain_system_or_die(const char *command);

The **explain_system_or_die** function is used to call the *system*(3) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_system*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int status = explain_system_or_die(command);
```

*command*
: The command, exactly as to be passed to the *system*(3) system call.

Returns: This function only returns on success, see *system*(3) for more information. On failure, prints an explanation and exits.

**explain_system_success_or_die**

void explain_system_success_or_die(const char *command);

The **explain_system_success_or_die** function is used to call the *system*(3) system call. On failure, including any exit status other than EXIT_SUCCESS, an explanation will be printed to *stderr*, obtained from *explain_system*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_system_success_or_die(command);
```

*command*
: The command, exactly as to be passed to the *system*(3) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

**explain_system_success**

int explain_system_success(const char *command);

The **explain_system_success** function is used to call the *system*(3) system call. On failure, including any exit status other than EXIT_SUCCESS, an explanation will be printed to *stderr*, obtained from *explain_system*(3). However, the priniting of an error message does **not** also cause *exit*(2) to be called.

This function is intended to be used in a fashion similar to the following example:

```
int status = explain_system_success(command);
```

*command*
: The command, exactly as to be passed to the *system*(3) system call.

Returns: the value returned by the *system*(3) system call. In all cases other than EXIT_SUCCESS, an error message will also have been printed to stderr.

## SEE ALSO

*system*(3)
: execute a shell command

*explain_system*(3)
explain *system*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tcdrain − explain *tcdrain*(3) errors

## SYNOPSIS

#include <libexplain/tcdrain.h>

const char *explain_tcdrain(int fildes);
const char *explain_errno_tcdrain(int errnum, int fildes);
void explain_message_tcdrain(char *message, int message_size, int fildes);
void explain_message_errno_tcdrain(char *message, int message_size, int errnum, int fildes);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tcdrain*(3) system call.

### explain_tcdrain

const char *explain_tcdrain(int fildes);

The **explain_tcdrain** function is used to obtain an explanation of an error returned by the *tcdrain*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *tcdrain*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcdrain(fildes) < 0)
{
   fprintf(stderr, "%s\n", explain_tcdrain(fildes));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcdrain_or_die*(3) function.

### explain_errno_tcdrain

const char *explain_errno_tcdrain(int errnum, int fildes);

The **explain_errno_tcdrain** function is used to obtain an explanation of an error returned by the *tcdrain*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcdrain*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcdrain(fildes) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_tcdrain(err, fildes));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcdrain_or_die*(3) function.

**explain_message_tcdrain**

void explain_message_tcdrain(char *message, int message_size, int fildes);

The **explain_message_tcdrain** function is used to obtain an explanation of an error returned by the *tcdrain*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *tcdrain*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcdrain(fildes) < 0)
{
    char message[3000];
    explain_message_tcdrain(message, sizeof(message), fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcdrain_or_die*(3) function.

**explain_message_errno_tcdrain**

void explain_message_errno_tcdrain(char *message, int message_size, int errnum, int fildes);

The **explain_message_errno_tcdrain** function is used to obtain an explanation of an error returned by the *tcdrain*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcdrain*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcdrain(fildes) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_tcdrain(message, sizeof(message), err,
    fildes);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcdrain_or_die*(3) function.

## SEE ALSO

*tcdrain*(3)
: Execute *tcdrain*(3)

*explain_tcdrain_or_die*(3)
: Execute *tcdrain*(3) and report errors

## COPYRIGHT

## NAME

explain_tcdrain_or_die − Execute *tcdrain*(3) and report errors

## SYNOPSIS

#include <libexplain/tcdrain.h>

void explain_tcdrain_or_die(int fildes);
int explain_tcdrain_on_error(int fildes);

## DESCRIPTION

The **explain_tcdrain_or_die** function is used to call the *tcdrain*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcdrain*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tcdrain_on_error** function is used to call the *tcdrain*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcdrain*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *tcdrain*(3) system call.

## RETURN VALUE

The **explain_tcdrain_or_die** function only returns on success, see *tcdrain*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tcdrain_on_error** function always returns the value return by the wrapped *tcdrain*(3) system call.

## EXAMPLE

The **explain_tcdrain_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_tcdrain_or_die(fildes);
```

## SEE ALSO

*tcdrain*(3)
Execute *tcdrain*(3)

*explain_tcdrain*(3)
explain *tcdrain*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tcflow − explain tcflow(3) errors

## SYNOPSIS

#include <libexplain/tcflow.h>

const char *explain_tcflow(int fildes, int action);
const char *explain_errno_tcflow(int errnum, int fildes, int action);
void explain_message_tcflow(char *message, int message_size, int fildes, int action);
void explain_message_errno_tcflow(char *message, int message_size, int errnum, int fildes, int action);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tcflow*(3) system call.

### explain_tcflow

const char *explain_tcflow(int fildes, int action);

The **explain_tcflow** function is used to obtain an explanation of an error returned by the *tcflow*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *tcflow*(3) system call.

*action* The original action, exactly as passed to the *tcflow*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcflow(fildes, action) < 0)
{
   fprintf(stderr, "%s\n", explain_tcflow(fildes, action));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcflow_or_die*(3) function.

### explain_errno_tcflow

const char *explain_errno_tcflow(int errnum, int fildes, int action);

The **explain_errno_tcflow** function is used to obtain an explanation of an error returned by the *tcflow*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcflow*(3) system call.

*action* The original action, exactly as passed to the *tcflow*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcflow(fildes, action) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_tcflow(err, fildes,
    action));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcflow_or_die*(3) function.

**explain_message_tcflow**

void explain_message_tcflow(char *message, int message_size, int fildes, int action);

The **explain_message_tcflow** function is used to obtain an explanation of an error returned by the *tcflow*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *tcflow*(3) system call.

*action* The original action, exactly as passed to the *tcflow*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcflow(fildes, action) < 0)
{
    char message[3000];
    explain_message_tcflow(message, sizeof(message), fildes,
    action);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcflow_or_die*(3) function.

**explain_message_errno_tcflow**

void explain_message_errno_tcflow(char *message, int message_size, int errnum, int fildes, int action);

The **explain_message_errno_tcflow** function is used to obtain an explanation of an error returned by the *tcflow*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcflow*(3) system call.

*action* The original action, exactly as passed to the *tcflow*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcflow(fildes, action) < 0)
{
```

```
    int err = errno;
    char message[3000];
   explain_message_errno_tcflow(message, sizeof(message), err,
   fildes, action);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcflow_or_die*(3) function.

## SEE ALSO

*tcflow*(3)
terminal flow control

*explain_tcflow_or_die*(3)
terminal flow control and report errors

## COPYRIGHT

## NAME

explain_tcflow_or_die − terminal flow control and report errors

## SYNOPSIS

#include <libexplain/tcflow.h>

void explain_tcflow_or_die(int fildes, int action);
int explain_tcflow_on_error(int fildes, int action);

## DESCRIPTION

The **explain_tcflow_or_die** function is used to call the *tcflow*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcflow*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tcflow_on_error** function is used to call the *tcflow*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcflow*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *tcflow*(3) system call.

*action* The action, exactly as to be passed to the *tcflow*(3) system call.

## RETURN VALUE

The **explain_tcflow_or_die** function only returns on success, see *tcflow*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tcflow_on_error** function always returns the value return by the wrapped *tcflow*(3) system call.

## EXAMPLE

The **explain_tcflow_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_tcflow_or_die(fildes, action);
```

## SEE ALSO

*tcflow*(3)
terminal flow control

*explain_tcflow*(3)
explain *tcflow*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tcflush − explain *tcflush*(3) errors

## SYNOPSIS

#include <libexplain/tcflush.h>

const char *explain_tcflush(int fildes, int selector);
const char *explain_errno_tcflush(int errnum, int fildes, int selector);
void explain_message_tcflush(char *message, int message_size, int fildes, int selector);
void explain_message_errno_tcflush(char *message, int message_size, int errnum, int fildes, int selector);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tcflush*(3) system call.

### explain_tcflush

const char *explain_tcflush(int fildes, int selector);

The **explain_tcflush** function is used to obtain an explanation of an error returned by the *tcflush*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *tcflush*(3) system call.

*selector* The original selector, exactly as passed to the *tcflush*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcflush(fildes, selector) < 0)
{
   fprintf(stderr, "%s\n", explain_tcflush(fildes, selector));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcflush_or_die*(3) function.

### explain_errno_tcflush

const char *explain_errno_tcflush(int errnum, int fildes, int selector);

The **explain_errno_tcflush** function is used to obtain an explanation of an error returned by the *tcflush*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcflush*(3) system call.

*selector* The original selector, exactly as passed to the *tcflush*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcflush(fildes, selector) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_tcflush(err, fildes,
    selector));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcflush_or_die*(3) function.

**explain_message_tcflush**

void explain_message_tcflush(char *message, int message_size, int fildes, int selector);

The **explain_message_tcflush** function is used to obtain an explanation of an error returned by the *tcflush*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *tcflush*(3) system call.

*selector* The original selector, exactly as passed to the *tcflush*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcflush(fildes, selector) < 0)
{
    char message[3000];
    explain_message_tcflush(message, sizeof(message), fildes,
    selector);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcflush_or_die*(3) function.

**explain_message_errno_tcflush**

void explain_message_errno_tcflush(char *message, int message_size, int errnum, int fildes, int selector);

The **explain_message_errno_tcflush** function is used to obtain an explanation of an error returned by the *tcflush*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcflush*(3) system call.

*selector* The original selector, exactly as passed to the *tcflush*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcflush(fildes, selector) < 0)
{
```

```
    int err = errno;
    char message[3000];
   explain_message_errno_tcflush(message, sizeof(message), err,
   fildes, selector);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcflush_or_die*(3) function.

## SEE ALSO

*tcflush*(3)
: discard terminal data

*explain_tcflush_or_die*(3)
: discard terminal data and report errors

## COPYRIGHT

## NAME

explain_tcflush_or_die − discard terminal data and report errors

## SYNOPSIS

#include <libexplain/tcflush.h>

void explain_tcflush_or_die(int fildes, int selector);
int explain_tcflush_on_error(int fildes, int selector);

## DESCRIPTION

The **explain_tcflush_or_die** function is used to call the *tcflush*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcflush*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tcflush_on_error** function is used to call the *tcflush*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcflush*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *tcflush*(3) system call.

*selector* The selector, exactly as to be passed to the *tcflush*(3) system call.

## RETURN VALUE

The **explain_tcflush_or_die** function only returns on success, see *tcflush*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tcflush_on_error** function always returns the value return by the wrapped *tcflush*(3) system call.

## EXAMPLE

The **explain_tcflush_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_tcflush_or_die(fildes, selector);
```

## SEE ALSO

*tcflush*(3)
discard terminal data

*explain_tcflush*(3)
explain *tcflush*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tcgetattr − explain *tcgetattr*(3) errors

## SYNOPSIS

#include <libexplain/tcgetattr.h>

const char *explain_tcgetattr(int fildes, struct termios *data);
const char *explain_errno_tcgetattr(int errnum, int fildes, struct termios *data);
void explain_message_tcgetattr(char *message, int message_size, int fildes, struct termios *data);
void explain_message_errno_tcgetattr(char *message, int message_size, int errnum, int fildes, struct termios *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tcgetattr*(3) system call.

### explain_tcgetattr

const char *explain_tcgetattr(int fildes, struct termios *data);

The **explain_tcgetattr** function is used to obtain an explanation of an error returned by the *tcgetattr*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *tcgetattr*(3) system call.

*data* The original data, exactly as passed to the *tcgetattr*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcgetattr(fildes, data) < 0)
{
    fprintf(stderr, "%s\n", explain_tcgetattr(fildes, data));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcgetattr_or_die*(3) function.

### explain_errno_tcgetattr

const char *explain_errno_tcgetattr(int errnum, int fildes, struct termios *data);

The **explain_errno_tcgetattr** function is used to obtain an explanation of an error returned by the *tcgetattr*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcgetattr*(3) system call.

*data* The original data, exactly as passed to the *tcgetattr*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcgetattr(fildes, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_tcgetattr(err, fildes,
   data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcgetattr_or_die*(3) function.

**explain_message_tcgetattr**

void explain_message_tcgetattr(char *message, int message_size, int fildes, struct termios *data);

The **explain_message_tcgetattr** function is used to obtain an explanation of an error returned by the *tcgetattr*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *tcgetattr*(3) system call.

*data* The original data, exactly as passed to the *tcgetattr*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcgetattr(fildes, data) < 0)
{
    char message[3000];
   explain_message_tcgetattr(message, sizeof(message), fildes,
   data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcgetattr_or_die*(3) function.

**explain_message_errno_tcgetattr**

void explain_message_errno_tcgetattr(char *message, int message_size, int errnum, int fildes, struct termios *data);

The **explain_message_errno_tcgetattr** function is used to obtain an explanation of an error returned by the *tcgetattr*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcgetattr*(3) system call.

*data* The original data, exactly as passed to the *tcgetattr*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcgetattr(fildes, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_tcgetattr(message, sizeof(message), err,
   fildes, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcgetattr_or_die*(3) function.

## SEE ALSO

*tcgetattr*(3)
get terminal parameters

*explain_tcgetattr_or_die*(3)
get terminal parameters and report errors

## COPYRIGHT

## NAME

explain_tcgetattr_or_die − get terminal parameters and report errors

## SYNOPSIS

#include <libexplain/tcgetattr.h>

void explain_tcgetattr_or_die(int fildes, struct termios *data);
int explain_tcgetattr_on_error(int fildes, struct termios *data);

## DESCRIPTION

The **explain_tcgetattr_or_die** function is used to call the *tcgetattr*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcgetattr*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tcgetattr_on_error** function is used to call the *tcgetattr*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcgetattr*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *tcgetattr*(3) system call.

*data* The data, exactly as to be passed to the *tcgetattr*(3) system call.

## RETURN VALUE

The **explain_tcgetattr_or_die** function only returns on success, see *tcgetattr*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tcgetattr_on_error** function always returns the value return by the wrapped *tcgetattr*(3) system call.

## EXAMPLE

The **explain_tcgetattr_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_tcgetattr_or_die(fildes, data);
```

## SEE ALSO

*tcgetattr*(3)
get terminal parameters

*explain_tcgetattr*(3)
explain *tcgetattr*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tcsendbreak − explain *tcsendbreak*(3) errors

## SYNOPSIS

#include <libexplain/tcsendbreak.h>

const char *explain_tcsendbreak(int fildes, int duration);
const char *explain_errno_tcsendbreak(int errnum, int fildes, int duration);
void explain_message_tcsendbreak(char *message, int message_size, int fildes, int duration);
void explain_message_errno_tcsendbreak(char *message, int message_size, int errnum, int fildes, int duration);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tcsendbreak*(3) system call.

### explain_tcsendbreak

const char *explain_tcsendbreak(int fildes, int duration);

The **explain_tcsendbreak** function is used to obtain an explanation of an error returned by the *tcsendbreak*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *tcsendbreak*(3) system call.

*duration* The original duration, exactly as passed to the *tcsendbreak*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcsendbreak(fildes, duration) < 0)
{
    fprintf(stderr, "%s\n", explain_tcsendbreak(fildes,
    duration));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcsendbreak_or_die*(3) function.

### explain_errno_tcsendbreak

const char *explain_errno_tcsendbreak(int errnum, int fildes, int duration);

The **explain_errno_tcsendbreak** function is used to obtain an explanation of an error returned by the *tcsendbreak*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcsendbreak*(3) system call.

*duration* The original duration, exactly as passed to the *tcsendbreak*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcsendbreak(fildes, duration) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_tcsendbreak(err, fildes,
   duration));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcsendbreak_or_die*(3) function.

**explain_message_tcsendbreak**

void explain_message_tcsendbreak(char *message, int message_size, int fildes, int duration);

The **explain_message_tcsendbreak** function is used to obtain an explanation of an error returned by the *tcsendbreak*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *tcsendbreak*(3) system call.

*duration* The original duration, exactly as passed to the *tcsendbreak*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcsendbreak(fildes, duration) < 0)
{
    char message[3000];
   explain_message_tcsendbreak(message, sizeof(message), fildes,
   duration);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcsendbreak_or_die*(3) function.

**explain_message_errno_tcsendbreak**

void explain_message_errno_tcsendbreak(char *message, int message_size, int errnum, int fildes, int duration);

The **explain_message_errno_tcsendbreak** function is used to obtain an explanation of an error returned by the *tcsendbreak*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcsendbreak*(3) system call.

*duration* The original duration, exactly as passed to the *tcsendbreak*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcsendbreak(fildes, duration) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_tcsendbreak(message, sizeof(message),
   err, fildes, duration);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcsendbreak_or_die*(3) function.

## SEE ALSO

*tcsendbreak*(3)
send terminal line break

*explain_tcsendbreak_or_die*(3)
send terminal line break and report errors

## COPYRIGHT

## NAME

explain_tcsendbreak_or_die − send terminal line break and report errors

## SYNOPSIS

#include <libexplain/tcsendbreak.h>

void explain_tcsendbreak_or_die(int fildes, int duration);
int explain_tcsendbreak_on_error(int fildes, int duration);

## DESCRIPTION

The **explain_tcsendbreak_or_die** function is used to call the *tcsendbreak*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcsendbreak*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tcsendbreak_on_error** function is used to call the *tcsendbreak*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcsendbreak*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *tcsendbreak*(3) system call.

*duration* The duration, exactly as to be passed to the *tcsendbreak*(3) system call.

## RETURN VALUE

The **explain_tcsendbreak_or_die** function only returns on success, see *tcsendbreak*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tcsendbreak_on_error** function always returns the value return by the wrapped *tcsendbreak*(3) system call.

## EXAMPLE

The **explain_tcsendbreak_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_tcsendbreak_or_die(fildes, duration);
```

## SEE ALSO

*tcsendbreak*(3)
send terminal line break

*explain_tcsendbreak*(3)
explain *tcsendbreak*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tcsetattr − explain *tcsetattr*(3) errors

## SYNOPSIS

#include <libexplain/tcsetattr.h>

const char *explain_tcsetattr(int fildes, int options, const struct termios *data);
const char *explain_errno_tcsetattr(int errnum, int fildes, int options, const struct termios *data);
void explain_message_tcsetattr(char *message, int message_size, int fildes, int options, const struct termios *data);
void explain_message_errno_tcsetattr(char *message, int message_size, int errnum, int fildes, int options, const struct termios *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tcsetattr*(3) system call.

### explain_tcsetattr

const char *explain_tcsetattr(int fildes, int options, const struct termios *data);

The **explain_tcsetattr** function is used to obtain an explanation of an error returned by the *tcsetattr*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *tcsetattr*(3) system call.

*options* The original options, exactly as passed to the *tcsetattr*(3) system call.

*data* The original data, exactly as passed to the *tcsetattr*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcsetattr(fildes, options, data) < 0)
{
    fprintf(stderr, "%s\n", explain_tcsetattr(fildes, options,
    data));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcsetattr_or_die*(3) function.

### explain_errno_tcsetattr

const char *explain_errno_tcsetattr(int errnum, int fildes, int options, const struct termios *data);

The **explain_errno_tcsetattr** function is used to obtain an explanation of an error returned by the *tcsetattr*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *tcsetattr*(3) system call.

*options* The original options, exactly as passed to the *tcsetattr*(3) system call.

*data* The original data, exactly as passed to the *tcsetattr*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcsetattr(fildes, options, data) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_tcsetattr(err, fildes,
   options, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcsetattr_or_die*(3) function.

**explain_message_tcsetattr**

void explain_message_tcsetattr(char *message, int message_size, int fildes, int options, const struct termios *data);

The **explain_message_tcsetattr** function is used to obtain an explanation of an error returned by the *tcsetattr*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *tcsetattr*(3) system call.

*options* The original options, exactly as passed to the *tcsetattr*(3) system call.

*data* The original data, exactly as passed to the *tcsetattr*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcsetattr(fildes, options, data) < 0)
{
    char message[3000];
   explain_message_tcsetattr(message, sizeof(message), fildes,
   options, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcsetattr_or_die*(3) function.

**explain_message_errno_tcsetattr**

void explain_message_errno_tcsetattr(char *message, int message_size, int errnum, int fildes, int options, const struct termios *data);

The **explain_message_errno_tcsetattr** function is used to obtain an explanation of an error returned by the *tcsetattr*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes*
: The original fildes, exactly as passed to the *tcsetattr*(3) system call.

*options*
: The original options, exactly as passed to the *tcsetattr*(3) system call.

*data*
: The original data, exactly as passed to the *tcsetattr*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (tcsetattr(fildes, options, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_tcsetattr(message, sizeof(message), err,
   fildes, options, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tcsetattr_or_die*(3) function.

## SEE ALSO

*tcsetattr*(3)
: set terminal attributes

*explain_tcsetattr_or_die*(3)
: set terminal attributes and report errors

## COPYRIGHT

## NAME

explain_tcsetattr_or_die − set terminal attributes and report errors

## SYNOPSIS

#include <libexplain/tcsetattr.h>

void explain_tcsetattr_or_die(int fildes, int options, const struct termios *data);
int explain_tcsetattr_on_error(int fildes, int options, const struct termios *data);

## DESCRIPTION

The **explain_tcsetattr_or_die** function is used to call the *tcsetattr*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcsetattr*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tcsetattr_on_error** function is used to call the *tcsetattr*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tcsetattr*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *tcsetattr*(3) system call.

*options* The options, exactly as to be passed to the *tcsetattr*(3) system call.

*data* The data, exactly as to be passed to the *tcsetattr*(3) system call.

## RETURN VALUE

The **explain_tcsetattr_or_die** function only returns on success, see *tcsetattr*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tcsetattr_on_error** function always returns the value return by the wrapped *tcsetattr*(3) system call.

## EXAMPLE

The **explain_tcsetattr_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_tcsetattr_or_die(fildes, options, data);
```

## SEE ALSO

*tcsetattr*(3)
set terminal attributes

*explain_tcsetattr*(3)
explain *tcsetattr*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_telldir − explain telldir(3) errors

## SYNOPSIS

#include <libexplain/telldir.h>

const char *explain_telldir(DIR *dir);
const char *explain_errno_telldir(int errnum, DIR *dir);
void explain_message_telldir(char *message, int message_size, DIR *dir);
void explain_message_errno_telldir(char *message, int message_size, int errnum, DIR *dir);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *telldir*(3) system call.

### explain_telldir

const char *explain_telldir(DIR *dir);

The **explain_telldir** function is used to obtain an explanation of an error returned by the *telldir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*dir* The original dir, exactly as passed to the *telldir*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
off_t result = telldir(dir);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_telldir(dir));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_telldir_or_die*(3) function.

### explain_errno_telldir

const char *explain_errno_telldir(int errnum, DIR *dir);

The **explain_errno_telldir** function is used to obtain an explanation of an error returned by the *telldir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *telldir*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
off_t result = telldir(dir);
```

```
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_telldir(err, dir));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_telldir_or_die*(3) function.

**explain_message_telldir**

void explain_message_telldir(char *message, int message_size, DIR *dir);

The **explain_message_telldir** function is used to obtain an explanation of an error returned by the *telldir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*dir* The original dir, exactly as passed to the *telldir*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
off_t result = telldir(dir);
if (result < 0)
{
    char message[3000];
   explain_message_telldir(message, sizeof(message), dir);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_telldir_or_die*(3) function.

**explain_message_errno_telldir**

void explain_message_errno_telldir(char *message, int message_size, int errnum, DIR *dir);

The **explain_message_errno_telldir** function is used to obtain an explanation of an error returned by the *telldir*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *telldir*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
off_t result = telldir(dir);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_telldir(message, sizeof(message), err,
```

```
                dir);
                 fprintf(stderr, "%s\n", message);
                 exit(EXIT_FAILURE);
            }
```

The above code example is available pre-packaged as the *explain_telldir_or_die*(3) function.

## SEE ALSO

*telldir*(3)
return current location in directory stream

*explain_telldir_or_die*(3)
return current location in directory stream and report errors

## COPYRIGHT

## NAME

explain_telldir_or_die − current location in directory and report errors

## SYNOPSIS

#include <libexplain/telldir.h>

off_t explain_telldir_or_die(DIR *dir);
off_t explain_telldir_on_error(DIR *dir);

## DESCRIPTION

The **explain_telldir_or_die** function is used to call the *telldir*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_telldir*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_telldir_on_error** function is used to call the *telldir*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_telldir*(3) function, but still returns to the caller.

*dir* The dir, exactly as to be passed to the *telldir*(3) system call.

## RETURN VALUE

The **explain_telldir_or_die** function only returns on success, see *telldir*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_telldir_on_error** function always returns the value return by the wrapped *telldir*(3) system call.

## EXAMPLE

The **explain_telldir_or_die** function is intended to be used in a fashion similar to the following example:

```
off_t result = explain_telldir_or_die(dir);
```

## SEE ALSO

*telldir*(3)
return current location in directory stream

*explain_telldir*(3)
explain *telldir*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tempnam − explain *tempnam*(3) errors

## SYNOPSIS

#include <libexplain/tempnam.h>

const char *explain_tempnam(const char *dir, const char *prefix);
const char *explain_errno_tempnam(int errnum, const char *dir, const char *prefix);
void explain_message_tempnam(char *message, int message_size, const char *dir, const char *prefix);
void explain_message_errno_tempnam(char *message, int message_size, int errnum, const char *dir, const char *prefix);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tempnam*(3) system call.

### explain_tempnam

const char *explain_tempnam(const char *dir, const char *prefix);

The **explain_tempnam** function is used to obtain an explanation of an error returned by the *tempnam*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*dir* The original dir, exactly as passed to the *tempnam*(3) system call.

*prefix* The original prefix, exactly as passed to the *tempnam*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = tempnam(dir, prefix);
if (!result)
{
   fprintf(stderr, "%s\n", explain_tempnam(dir, prefix));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tempnam_or_die*(3) function.

### explain_errno_tempnam

const char *explain_errno_tempnam(int errnum, const char *dir, const char *prefix);

The **explain_errno_tempnam** function is used to obtain an explanation of an error returned by the *tempnam*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *tempnam*(3) system call.

*prefix* The original prefix, exactly as passed to the *tempnam*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other

functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = tempnam(dir, prefix);
if (!result)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_tempnam(err, dir,
   prefix));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tempnam_or_die*(3) function.

**explain_message_tempnam**

void explain_message_tempnam(char *message, int message_size, const char *dir, const char *prefix);

The **explain_message_tempnam** function is used to obtain an explanation of an error returned by the *tempnam*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*dir* The original dir, exactly as passed to the *tempnam*(3) system call.

*prefix* The original prefix, exactly as passed to the *tempnam*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = tempnam(dir, prefix);
if (!result)
{
    char message[3000];
   explain_message_tempnam(message, sizeof(message), dir,
   prefix);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tempnam_or_die*(3) function.

**explain_message_errno_tempnam**

void explain_message_errno_tempnam(char *message, int message_size, int errnum, const char *dir, const char *prefix);

The **explain_message_errno_tempnam** function is used to obtain an explanation of an error returned by the *tempnam*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dir* The original dir, exactly as passed to the *tempnam*(3) system call.

*prefix* The original prefix, exactly as passed to the *tempnam*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = tempnam(dir, prefix);
if (!result)
{
    int err = errno;
    char message[3000];
   explain_message_errno_tempnam(message, sizeof(message), err,
   dir, prefix);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tempnam_or_die*(3) function.

## SEE ALSO

*tempnam*(3)
create a name for a temporary file

*explain_tempnam_or_die*(3)
create a name for a temporary file and report errors

## COPYRIGHT

## NAME

explain_tempnam_or_die − create a name for a temporary file and report errors

## SYNOPSIS

#include <libexplain/tempnam.h>

char *explain_tempnam_or_die(const char *dir, const char *prefix);
char *explain_tempnam_on_error(const char *dir, const char *prefix);

## DESCRIPTION

The **explain_tempnam_or_die** function is used to call the *tempnam*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tempnam*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tempnam_on_error** function is used to call the *tempnam*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tempnam*(3) function, but still returns to the caller.

*dir* The dir, exactly as to be passed to the *tempnam*(3) system call.

*prefix* The prefix, exactly as to be passed to the *tempnam*(3) system call.

## RETURN VALUE

The **explain_tempnam_or_die** function only returns on success, see *tempnam*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tempnam_on_error** function always returns the value return by the wrapped *tempnam*(3) system call.

## EXAMPLE

The **explain_tempnam_or_die** function is intended to be used in a fashion similar to the following example:

```
char *result = explain_tempnam_or_die(dir, prefix);
```

## SEE ALSO

*tempnam*(3)
create a name for a temporary file

*explain_tempnam*(3)
explain *tempnam*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_time − explain time(2) errors

## SYNOPSIS

#include <libexplain/time.h>

const char *explain_time(time_t *t);
const char *explain_errno_time(int errnum, time_t *t);
void explain_message_time(char *message, int message_size, time_t *t);
void explain_message_errno_time(char *message, int message_size, int errnum, time_t *t);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *time*(2) system call.

### explain_time

const char *explain_time(time_t *t);

The **explain_time** function is used to obtain an explanation of an error returned by the *time*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*t* The original t, exactly as passed to the *time*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
time_t result = time(t);
if (result == (time_t)-1)
{
   fprintf(stderr, "%s\n", explain_time(t));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_time_or_die*(3) function.

### explain_errno_time

const char *explain_errno_time(int errnum, time_t *t);

The **explain_errno_time** function is used to obtain an explanation of an error returned by the *time*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*t* The original t, exactly as passed to the *time*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
time_t result = time(t);
```

```
if (result == (time_t)-1)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_time(err, t));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_time_or_die*(3) function.

**explain_message_time**

void explain_message_time(char *message, int message_size, time_t *t);

The **explain_message_time** function is used to obtain an explanation of an error returned by the *time*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*t* The original t, exactly as passed to the *time*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
time_t result = time(t);
if (result == (time_t)-1)
{
    char message[3000];
   explain_message_time(message, sizeof(message), t);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_time_or_die*(3) function.

**explain_message_errno_time**

void explain_message_errno_time(char *message, int message_size, int errnum, time_t *t);

The **explain_message_errno_time** function is used to obtain an explanation of an error returned by the *time*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*t* The original t, exactly as passed to the *time*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
time_t result = time(t);
if (result == (time_t)-1)
{
    int err = errno;
    char message[3000];
   explain_message_errno_time(message, sizeof(message), err, t);
```

```
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_time_or_die*(3) function.

## SEE ALSO

*time*(2) get time in seconds

*explain_time_or_die*(3)
get time in seconds and report errors

## COPYRIGHT

## NAME

explain_time_or_die − get time in seconds and report errors

## SYNOPSIS

#include <libexplain/time.h>

time_t explain_time_or_die(time_t *t);
time_t explain_time_on_error(time_t *t);

## DESCRIPTION

The **explain_time_or_die** function is used to call the *time*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_time*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_time_on_error** function is used to call the *time*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_time*(3) function, but still returns to the caller.

*t* The t, exactly as to be passed to the *time*(2) system call.

## RETURN VALUE

The **explain_time_or_die** function only returns on success, see *time*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_time_on_error** function always returns the value return by the wrapped *time*(2) system call.

## EXAMPLE

The **explain_time_or_die** function is intended to be used in a fashion similar to the following example:

```
time_t result = explain_time_or_die(t);
```

## SEE ALSO

*time*(2) get time in seconds

*explain_time*(3)
explain *time*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_timerfd_create − explain timerfd_create(2) errors

## SYNOPSIS

#include <libexplain/timerfd_create.h>

const char *explain_timerfd_create(int clockid, int flags);
const char *explain_errno_timerfd_create(int errnum, int clockid, int flags);
void explain_message_timerfd_create(char *message, int message_size, int clockid, int flags);
void explain_message_errno_timerfd_create(char *message, int message_size, int errnum, int clockid, int flags);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *timerfd_create*(2) system call.

### explain_timerfd_create

const char *explain_timerfd_create(int clockid, int flags);

The **explain_timerfd_create** function is used to obtain an explanation of an error returned by the *timerfd_create*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*clockid* The original clockid, exactly as passed to the *timerfd_create*(2) system call.

*flags* The original flags, exactly as passed to the *timerfd_create*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = timerfd_create(clockid, flags);
if (result < 0)
{
    fprintf(stderr, "%s\n", explain_timerfd_create(clockid,
    flags));
     exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_timerfd_create_or_die*(3) function.

### explain_errno_timerfd_create

const char *explain_errno_timerfd_create(int errnum, int clockid, int flags);

The **explain_errno_timerfd_create** function is used to obtain an explanation of an error returned by the *timerfd_create*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*clockid* The original clockid, exactly as passed to the *timerfd_create*(2) system call.

*flags* The original flags, exactly as passed to the *timerfd_create*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = timerfd_create(clockid, flags);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_timerfd_create(err,
   clockid, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_timerfd_create_or_die*(3) function.

**explain_message_timerfd_create**

void explain_message_timerfd_create(char *message, int message_size, int clockid, int flags);

The **explain_message_timerfd_create** function is used to obtain an explanation of an error returned by the *timerfd_create*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*clockid* The original clockid, exactly as passed to the *timerfd_create*(2) system call.

*flags* The original flags, exactly as passed to the *timerfd_create*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = timerfd_create(clockid, flags);
if (result < 0)
{
    char message[3000];
   explain_message_timerfd_create(message, sizeof(message),
   clockid, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_timerfd_create_or_die*(3) function.

**explain_message_errno_timerfd_create**

void explain_message_errno_timerfd_create(char *message, int message_size, int errnum, int clockid, int flags);

The **explain_message_errno_timerfd_create** function is used to obtain an explanation of an error returned by the *timerfd_create*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*clockid* The original clockid, exactly as passed to the *timerfd_create*(2) system call.

*flags* The original flags, exactly as passed to the *timerfd_create*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
int result = timerfd_create(clockid, flags);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_timerfd_create(message, sizeof(message),
   err, clockid, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_timerfd_create_or_die*(3) function.

## SEE ALSO

*timerfd_create*(2)
timers that notify via file descriptors

*explain_timerfd_create_or_die*(3)
timers that notify via file descriptors and report errors

## COPYRIGHT

## NAME

explain_timerfd_create_or_die − create file descriptor timers and report errors

## SYNOPSIS

#include <libexplain/timerfd_create.h>

int explain_timerfd_create_or_die(int clockid, int flags);
int explain_timerfd_create_on_error(int clockid, int flags);

## DESCRIPTION

The **explain_timerfd_create_or_die** function is used to call the *timerfd_create*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_timerfd_create*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_timerfd_create_on_error** function is used to call the *timerfd_create*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_timerfd_create*(3) function, but still returns to the caller.

*clockid* The clockid, exactly as to be passed to the *timerfd_create*(2) system call.

*flags* The flags, exactly as to be passed to the *timerfd_create*(2) system call.

## RETURN VALUE

The **explain_timerfd_create_or_die** function only returns on success, see *timerfd_create*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_timerfd_create_on_error** function always returns the value return by the wrapped *timerfd_create*(2) system call.

## EXAMPLE

The **explain_timerfd_create_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_timerfd_create_or_die(clockid, flags);
```

## SEE ALSO

*timerfd_create*(2)
timers that notify via file descriptors

*explain_timerfd_create*(3)
explain *timerfd_create*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tmpfile − explain *tmpfile*(3) errors

## SYNOPSIS

#include <libexplain/tmpfile.h>

const char *explain_tmpfile(void);
const char *explain_errno_tmpfile(int errnum, void);
void explain_message_tmpfile(char *message, int message_size, void);
void explain_message_errno_tmpfile(char *message, int message_size, int errnum, void);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tmpfile*(3) system call.

### explain_tmpfile

const char *explain_tmpfile(void);

The **explain_tmpfile** function is used to obtain an explanation of an error returned by the *tmpfile*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
FILE *result = tmpfile();
if (!result)
{
   fprintf(stderr, "%s\n", explain_tmpfile());
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tmpfile_or_die*(3) function.

### explain_errno_tmpfile

const char *explain_errno_tmpfile(int errnum, void);

The **explain_errno_tmpfile** function is used to obtain an explanation of an error returned by the *tmpfile*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
FILE *result = tmpfile();
if (!result)
{
    int err = errno;
```

```
    fprintf(stderr, "%s\n", explain_errno_tmpfile(err, ));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tmpfile_or_die*(3) function.

**explain_message_tmpfile**

void explain_message_tmpfile(char *message, int message_size, void);

The **explain_message_tmpfile** function is used to obtain an explanation of an error returned by the *tmpfile*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
FILE *result = tmpfile();
if (!result)
{
    char message[3000];
    explain_message_tmpfile(message, sizeof(message), );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tmpfile_or_die*(3) function.

**explain_message_errno_tmpfile**

void explain_message_errno_tmpfile(char *message, int message_size, int errnum, void);

The **explain_message_errno_tmpfile** function is used to obtain an explanation of an error returned by the *tmpfile*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
FILE *result = tmpfile();
if (!result)
{
    int err = errno;
    char message[3000];
    explain_message_errno_tmpfile(message, sizeof(message), err,
    );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tmpfile_or_die*(3) function.

## SEE ALSO

*tmpfile*(3)
: create a temporary file

*explain_tmpfile_or_die*(3)
: create a temporary file and report errors

## COPYRIGHT

## NAME

explain_tmpfile_or_die − create a temporary file and report errors

## SYNOPSIS

#include <libexplain/tmpfile.h>

FILE *explain_tmpfile_or_die(void);
FILE *explain_tmpfile_on_error(void);

## DESCRIPTION

The **explain_tmpfile_or_die** function is used to call the *tmpfile*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tmpfile*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tmpfile_on_error** function is used to call the *tmpfile*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tmpfile*(3) function, but still returns to the caller.

## RETURN VALUE

The **explain_tmpfile_or_die** function only returns on success, see *tmpfile*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tmpfile_on_error** function always returns the value return by the wrapped *tmpfile*(3) system call.

## EXAMPLE

The **explain_tmpfile_or_die** function is intended to be used in a fashion similar to the following example:

```
FILE *result = explain_tmpfile_or_die();
```

## SEE ALSO

*tmpfile*(3)
: create a temporary file

*explain_tmpfile*(3)
: explain *tmpfile*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_tmpnam − explain *tmpnam*(3) errors

## SYNOPSIS

#include <libexplain/tmpnam.h>

const char *explain_tmpnam(char *pathname);
const char *explain_errno_tmpnam(int errnum, char *pathname);
void explain_message_tmpnam(char *message, int message_size, char *pathname);
void explain_message_errno_tmpnam(char *message, int message_size, int errnum, char *pathname);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *tmpnam*(3) system call.

### explain_tmpnam

const char *explain_tmpnam(char *pathname);

The **explain_tmpnam** function is used to obtain an explanation of an error returned by the *tmpnam*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *tmpnam*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = tmpnam(pathname);
if (!result)
{
   fprintf(stderr, "%s\n", explain_tmpnam(pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tmpnam_or_die*(3) function.

### explain_errno_tmpnam

const char *explain_errno_tmpnam(int errnum, char *pathname);

The **explain_errno_tmpnam** function is used to obtain an explanation of an error returned by the *tmpnam*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *tmpnam*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = tmpnam(pathname);
if (!result)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_tmpnam(err, pathname));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tmpnam_or_die*(3) function.

**explain_message_tmpnam**

void explain_message_tmpnam(char *message, int message_size, char *pathname);

The **explain_message_tmpnam** function is used to obtain an explanation of an error returned by the *tmpnam*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *tmpnam*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = tmpnam(pathname);
if (!result)
{
    char message[3000];
   explain_message_tmpnam(message, sizeof(message), pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tmpnam_or_die*(3) function.

**explain_message_errno_tmpnam**

void explain_message_errno_tmpnam(char *message, int message_size, int errnum, char *pathname);

The **explain_message_errno_tmpnam** function is used to obtain an explanation of an error returned by the *tmpnam*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *tmpnam*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
char *result = tmpnam(pathname);
if (!result)
```

```
{
    int err = errno;
    char message[3000];
   explain_message_errno_tmpnam(message, sizeof(message), err,
   pathname);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_tmpnam_or_die*(3) function.

## SEE ALSO

*tmpnam*(3)
: create a name for a temporary file

*explain_tmpnam_or_die*(3)
: create a name for a temporary file and report errors

## COPYRIGHT

## NAME

explain_tmpnam_or_die − create a name for a temporary file and report errors

## SYNOPSIS

#include <libexplain/tmpnam.h>

char *explain_tmpnam_or_die(char *pathname);
char *explain_tmpnam_on_error(char *pathname);

## DESCRIPTION

The **explain_tmpnam_or_die** function is used to call the *tmpnam*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tmpnam*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_tmpnam_on_error** function is used to call the *tmpnam*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_tmpnam*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *tmpnam*(3) system call.

## RETURN VALUE

The **explain_tmpnam_or_die** function only returns on success, see *tmpnam*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_tmpnam_on_error** function always returns the value return by the wrapped *tmpnam*(3) system call.

## EXAMPLE

The **explain_tmpnam_or_die** function is intended to be used in a fashion similar to the following example:

```
char *result = explain_tmpnam_or_die(pathname);
```

## SEE ALLOW

*tmpnam*(3)
: create a name for a temporary file

*explain_tmpnam*(3)
: explain *tmpnam*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_truncate − explain truncate(2) errors

## SYNOPSIS

#include <libexplain/truncate.h>

const char *explain_truncate(const char *pathname, long long length);
const char *explain_errno_truncate(int errnum, const char *pathname, long long length);
void explain_message_truncate(char *message, int message_size, const char *pathname, long long length);
void explain_message_errno_truncate(char *message, int message_size, int errnum, const char *pathname, long long length);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *truncate*(2) system call.

### explain_truncate

const char *explain_truncate(const char *pathname, long long length);

The **explain_truncate** function is used to obtain an explanation of an error returned by the *truncate*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (truncate(pathname, length) < 0)
{
    fprintf(stderr, "%s\n", explain_truncate(pathname, length));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *truncate*(2) system call.

*length*
: The original length, exactly as passed to the *truncate*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_truncate

const char *explain_errno_truncate(int errnum, const char *pathname, long long length);

The **explain_errno_truncate** function is used to obtain an explanation of an error returned by the *truncate*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (truncate(pathname, length) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_truncate(err, pathname, length));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *truncate*(2) system call.

*length* The original length, exactly as passed to the *truncate*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_truncate**

void explain_message_truncate(char *message, int message_size, const char *pathname, long long length);

The **explain_message_truncate** function may be used to obtain an explanation of an error returned by the *truncate*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (truncate(pathname, length) < 0)
{
    char message[3000];
    explain_message_truncate(message, sizeof(message), pathname, length);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *truncate*(2) system call.

*length* The original length, exactly as passed to the *truncate*(2) system call.

**explain_message_errno_truncate**

void explain_message_errno_truncate(char *message, int message_size, int errnum, const char *pathname, long long length);

The **explain_message_errno_truncate** function may be used to obtain an explanation of an error returned by the *truncate*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (truncate(pathname, length) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_truncate(message, sizeof(message), err,
        pathname, length);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *truncate*(2) system call.

*length* The original length, exactly as passed to the *truncate*(2) system call.

## SEE ALSO

*truncate*(2)
truncate a file to a specified length

*explain_truncate_or_die*(3)
truncate a file to a specified length and report errors

## COPYRIGHT

## NAME

explain_truncate_or_die − truncate a file and report errors

## SYNOPSIS

#include <libexplain/truncate.h>

void explain_truncate_or_die(const char *pathname, long long length);

## DESCRIPTION

The **explain_truncate_or_die** function is used to call the *truncate*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_truncate*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_truncate_or_die(pathname, length);
```

*pathname*
: The pathname, exactly as to be passed to the *truncate*(2) system call.

*length*
: The length, exactly as to be passed to the *truncate*(2) system call.

Returns:
: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*truncate*(2)
: truncate a file to a specified length

*explain_truncate*(3)
: explain *truncate*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_ungetc − explain *ungetc*(3) errors

## SYNOPSIS

#include <libexplain/ungetc.h>

const char *explain_ungetc(int c, FILE *fp);
const char *explain_errno_ungetc(int errnum, int c, FILE *fp);
void explain_message_ungetc(char *message, int message_size, int c, FILE *fp);
void explain_message_errno_ungetc(char *message, int message_size, int errnum, int c, FILE *fp);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *ungetc*(3) system call.

### explain_ungetc

const char *explain_ungetc(int c, FILE *fp);

The **explain_ungetc** function is used to obtain an explanation of an error returned by the *ungetc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*c* The original c, exactly as passed to the *ungetc*(3) system call.

*fp* The original fp, exactly as passed to the *ungetc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ungetc(c, fp) < 0)
{
   fprintf(stderr, "%s\n", explain_ungetc(c, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ungetc_or_die*(3) function.

### explain_errno_ungetc

const char *explain_errno_ungetc(int errnum, int c, FILE *fp);

The **explain_errno_ungetc** function is used to obtain an explanation of an error returned by the *ungetc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*c* The original c, exactly as passed to the *ungetc*(3) system call.

*fp* The original fp, exactly as passed to the *ungetc*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ungetc(c, fp) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_ungetc(err, c, fp));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ungetc_or_die*(3) function.

**explain_message_ungetc**

void explain_message_ungetc(char *message, int message_size, int c, FILE *fp);

The **explain_message_ungetc** function is used to obtain an explanation of an error returned by the *ungetc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*c* The original c, exactly as passed to the *ungetc*(3) system call.

*fp* The original fp, exactly as passed to the *ungetc*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ungetc(c, fp) < 0)
{
    char message[3000];
    explain_message_ungetc(message, sizeof(message), c, fp);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ungetc_or_die*(3) function.

**explain_message_errno_ungetc**

void explain_message_errno_ungetc(char *message, int message_size, int errnum, int c, FILE *fp);

The **explain_message_errno_ungetc** function is used to obtain an explanation of an error returned by the *ungetc*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*c* The original c, exactly as passed to the *ungetc*(3) system call.

*fp* The original fp, exactly as passed to the *ungetc*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ungetc(c, fp) < 0)
{
    int err = errno;
    char message[3000];
```

```
        explain_message_errno_ungetc(message, sizeof(message), err, c,
        fp);
         fprintf(stderr, "%s\n", message);
         exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_ungetc_or_die*(3) function.

## SEE ALSO

*ungetc*(3)
push a character back to a stream

*explain_ungetc_or_die*(3)
push a character back to a stream and report errors

## COPYRIGHT

## NAME

explain_ungetc_or_die − push a character back to a stream and report errors

## SYNOPSIS

#include <libexplain/ungetc.h>

void explain_ungetc_or_die(int c, FILE *fp);
int explain_ungetc_on_error(int c, FILE *fp);

## DESCRIPTION

The **explain_ungetc_or_die** function is used to call the *ungetc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ungetc*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_ungetc_on_error** function is used to call the *ungetc*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ungetc*(3) function, but still returns to the caller.

*c* The c, exactly as to be passed to the *ungetc*(3) system call.

*fp* The fp, exactly as to be passed to the *ungetc*(3) system call.

## RETURN VALUE

The **explain_ungetc_or_die** function only returns on success, see *ungetc*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_ungetc_on_error** function always returns the value return by the wrapped *ungetc*(3) system call.

## EXAMPLE

The **explain_ungetc_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_ungetc_or_die(c, fp);
```

## SEE ALSO

*ungetc*(3)
push a character back to a stream

*explain_ungetc*(3)
explain *ungetc*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_unlink − explain unlink(2) errors

## SYNOPSIS

```
#include <libexplain/unlink.h>
const char *explain_unlink(const char *pathname);
void explain_message_unlink(char *message, int message_size, const char *pathname);
const char *explain_errno_unlink(int errnum, const char *pathname);
void explain_message_errno_unlink(char *message, int message_size, int errnum, const char *pathname);
```

## DESCRIPTION

These functions may be used to obtain explanations for *unlink*(2) errors.

### explain_unlink

const char *explain_unlink(const char *pathname);

The explain_unlink function is used to obtain an explanation of an error returned by the *unlink*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (unlink(pathname) < 0)
{
    fprintf(stderr, '%s0, explain_unlink(pathname));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *unlink*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_unlink

const char *explain_errno_unlink(int errnum, const char * pathname);

The explain_errno_unlink function is used to obtain an explanation of an error returned by the *unlink*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (unlink(pathname) < 0)
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_unlink(err, pathname));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *unlink*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_unlink**

void explain_message_unlink(char *message, int message_size, const char *pathname);

The explain_message_unlink function is used to obtain an explanation of an error returned by the *unlink*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (unlink(pathname) < 0)
{
    char message[3000];
    explain_message_unlink(message, sizeof(message), pathname);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *unlink*(2) system call.

**explain_message_errno_unlink**

void explain_message_errno_unlink(char *message, int message_size, int errnum, const char *pathname);

The explain_message_errno_unlink function is used to obtain an explanation of an error returned by the *unlink*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (unlink(pathname) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_unlink(message, sizeof(message), err,
        pathname);
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *unlink*(2) system call.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_unlink_or_die − delete a file and report errors

## SYNOPSIS

#include <libexplain/unlink.h>

void explain_unlink_or_die(const char *pathname);

## DESCRIPTION

The **explain_unlink_or_die** function is used to call the *unlink*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_unlink*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_unlink_or_die(pathname);
```

*pathname*
: The pathname, exactly as to be passed to the *unlink*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*unlink*(2)
: delete a name and possibly the file it refers to

*explain_unlink*(3)
: explain *unlink*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_unsetenv − explain *unsetenv*(3) errors

## SYNOPSIS

#include <libexplain/unsetenv.h>

const char *explain_unsetenv(const char *name);
const char *explain_errno_unsetenv(int errnum, const char *name);
void explain_message_unsetenv(char *message, int message_size, const char *name);
void explain_message_errno_unsetenv(char *message, int message_size, int errnum, const char *name);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *unsetenv*(3) system call.

### explain_unsetenv

const char *explain_unsetenv(const char *name);

The **explain_unsetenv** function is used to obtain an explanation of an error returned by the *unsetenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*name* The original name, exactly as passed to the *unsetenv*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (unsetenv(name) < 0)
{
   fprintf(stderr, "%s\n", explain_unsetenv(name));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_unsetenv_or_die*(3) function.

### explain_errno_unsetenv

const char *explain_errno_unsetenv(int errnum, const char *name);

The **explain_errno_unsetenv** function is used to obtain an explanation of an error returned by the *unsetenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*name* The original name, exactly as passed to the *unsetenv*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (unsetenv(name) < 0)
{
```

```
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_unsetenv(err, name));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_unsetenv_or_die*(3) function.

**explain_message_unsetenv**

void explain_message_unsetenv(char *message, int message_size, const char *name);

The **explain_message_unsetenv** function is used to obtain an explanation of an error returned by the *unsetenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*name* The original name, exactly as passed to the *unsetenv*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (unsetenv(name) < 0)
{
    char message[3000];
    explain_message_unsetenv(message, sizeof(message), name);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_unsetenv_or_die*(3) function.

**explain_message_errno_unsetenv**

void explain_message_errno_unsetenv(char *message, int message_size, int errnum, const char *name);

The **explain_message_errno_unsetenv** function is used to obtain an explanation of an error returned by the *unsetenv*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*name* The original name, exactly as passed to the *unsetenv*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (unsetenv(name) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_unsetenv(message, sizeof(message), err,
    name);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_unsetenv_or_die*(3) function.

## SEE ALSO

*unsetenv*(3)
change or add an environment variable

*explain_unsetenv_or_die*(3)
change or add an environment variable and report errors

## COPYRIGHT

## NAME

explain_unsetenv_or_die − remove an environment variable and report errors

## SYNOPSIS

#include <libexplain/unsetenv.h>

void explain_unsetenv_or_die(const char *name);
int explain_unsetenv_on_error(const char *name);

## DESCRIPTION

The **explain_unsetenv_or_die** function is used to call the *unsetenv*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_unsetenv*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_unsetenv_on_error** function is used to call the *unsetenv*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_unsetenv*(3) function, but still returns to the caller.

*name* The name, exactly as to be passed to the *unsetenv*(3) system call.

## RETURN VALUE

The **explain_unsetenv_or_die** function only returns on success, see *unsetenv*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_unsetenv_on_error** function always returns the value return by the wrapped *unsetenv*(3) system call.

## EXAMPLE

The **explain_unsetenv_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_unsetenv_or_die(name);
```

## SEE ALSO

*unsetenv*(3)
change or add an environment variable

*explain_unsetenv*(3)
explain *unsetenv*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_ustat − explain ustat(2) errors

## SYNOPSIS

#include <libexplain/ustat.h>

const char *explain_ustat(dev_t dev, struct ustat *ubuf);
const char *explain_errno_ustat(int errnum, dev_t dev, struct ustat *ubuf);
void explain_message_ustat(char *message, int message_size, dev_t dev, struct ustat *ubuf);
void explain_message_errno_ustat(char *message, int message_size, int errnum, dev_t dev, struct ustat *ubuf);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *ustat*(2) system call.

### explain_ustat

const char *explain_ustat(dev_t dev, struct ustat *ubuf);

The **explain_ustat** function is used to obtain an explanation of an error returned by the *ustat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*dev* The original dev, exactly as passed to the *ustat*(2) system call.

*ubuf* The original ubuf, exactly as passed to the *ustat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ustat(dev, ubuf) < 0)
{
   fprintf(stderr, "%s\n", explain_ustat(dev, ubuf));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ustat_or_die*(3) function.

### explain_errno_ustat

const char *explain_errno_ustat(int errnum, dev_t dev, struct ustat *ubuf);

The **explain_errno_ustat** function is used to obtain an explanation of an error returned by the *ustat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dev* The original dev, exactly as passed to the *ustat*(2) system call.

*ubuf* The original ubuf, exactly as passed to the *ustat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ustat(dev, ubuf) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_ustat(err, dev, ubuf));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ustat_or_die*(3) function.

**explain_message_ustat**

void explain_message_ustat(char *message, int message_size, dev_t dev, struct ustat *ubuf);

The **explain_message_ustat** function is used to obtain an explanation of an error returned by the *ustat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*dev* The original dev, exactly as passed to the *ustat*(2) system call.

*ubuf* The original ubuf, exactly as passed to the *ustat*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ustat(dev, ubuf) < 0)
{
    char message[3000];
    explain_message_ustat(message, sizeof(message), dev, ubuf);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ustat_or_die*(3) function.

**explain_message_errno_ustat**

void explain_message_errno_ustat(char *message, int message_size, int errnum, dev_t dev, struct ustat *ubuf);

The **explain_message_errno_ustat** function is used to obtain an explanation of an error returned by the *ustat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*dev* The original dev, exactly as passed to the *ustat*(2) system call.

*ubuf* The original ubuf, exactly as passed to the *ustat*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (ustat(dev, ubuf) < 0)
{
```

```
    int err = errno;
    char message[3000];
    explain_message_errno_ustat(message, sizeof(message), err,
    dev, ubuf);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_ustat_or_die*(3) function.

## SEE ALSO

*ustat*(2) get file system statistics

*explain_ustat_or_die*(3)
get file system statistics and report errors

## COPYRIGHT

## NAME

explain_ustat_or_die − get file system statistics and report errors

## SYNOPSIS

#include <libexplain/ustat.h>

void explain_ustat_or_die(dev_t dev, struct ustat *ubuf);
int explain_ustat_on_error(dev_t dev, struct ustat *ubuf);

## DESCRIPTION

The **explain_ustat_or_die** function is used to call the *ustat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ustat*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_ustat_on_error** function is used to call the *ustat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_ustat*(3) function, but still returns to the caller.

*dev* The dev, exactly as to be passed to the *ustat*(2) system call.

*ubuf* The ubuf, exactly as to be passed to the *ustat*(2) system call.

## RETURN VALUE

The **explain_ustat_or_die** function only returns on success, see *ustat*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_ustat_on_error** function always returns the value return by the wrapped *ustat*(2) system call.

## EXAMPLE

The **explain_ustat_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_ustat_or_die(dev, ubuf);
```

## SEE ALSO

*ustat*(2) get file system statistics

*explain_ustat*(3)
explain *ustat*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_utime − explain utime(2) errors

## SYNOPSIS

#include <libexplain/utime.h>

const char *explain_utime(const char *pathname, const struct utimbuf *times);
const char *explain_errno_utime(int errnum, const char *pathname, const struct utimbuf *times);
void explain_message_utime(char *message, int message_size, const char *pathname, const struct utimbuf *times);
void explain_message_errno_utime(char *message, int message_size, int errnum, const char *pathname, const struct utimbuf *times);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *utime*(2) system call.

### explain_utime

const char *explain_utime(const char *pathname, const struct utimbuf *times);

The **explain_utime** function is used to obtain an explanation of an error returned by the *utime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (utime(pathname, times) < 0)
{
    fprintf(stderr, "%s\n", explain_utime(pathname, times));
    exit(EXIT_FAILURE);
}
```

*pathname*
: The original pathname, exactly as passed to the *utime*(2) system call.

*times*
: The original times, exactly as passed to the *utime*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_utime

const char *explain_errno_utime(int errnum, const char *pathname, const struct utimbuf *times);

The **explain_errno_utime** function is used to obtain an explanation of an error returned by the *utime*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (utime(pathname, times) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_utime(err, pathname, times));
    exit(EXIT_FAILURE);
}
```

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *utime*(2) system call.

*times* The original times, exactly as passed to the *utime*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_utime**

void explain_message_utime(char *message, int message_size, const char *pathname, const struct utimbuf *times);

The **explain_message_utime** function may be used to obtain an explanation of an error returned by the *utime*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (utime(pathname, times) < 0)
{
    char message[3000];
    explain_message_utime(message, sizeof(message), pathname, times);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *utime*(2) system call.

*times* The original times, exactly as passed to the *utime*(2) system call.

**explain_message_errno_utime**

void explain_message_errno_utime(char *message, int message_size, int errnum, const char *pathname, const struct utimbuf *times);

The **explain_message_errno_utime** function may be used to obtain an explanation of an error returned by the *utime*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (utime(pathname, times) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_utime(message, sizeof(message), err,
        pathname, times);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *utime*(2) system call.

*times* The original times, exactly as passed to the *utime*(2) system call.

## SEE ALSO

*utime*(2) change file last access and modification times

*explain_utime_or_die*(3)
change file last access and modification times and report errors

## COPYRIGHT

## NAME

explain_utimens − explain *utimens*(2) errors

## SYNOPSIS

#include <libexplain/utimens.h>

const char *explain_utimens(const char *pathname, const struct timespec *data);
const char *explain_errno_utimens(int errnum, const char *pathname, const struct timespec *data);
void explain_message_utimens(char *message, int message_size, const char *pathname, const struct timespec *data);
void explain_message_errno_utimens(char *message, int message_size, int errnum, const char *pathname, const struct timespec *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *utimens*(2) system call.

### explain_utimens

const char *explain_utimens(const char *pathname, const struct timespec *data);

The **explain_utimens** function is used to obtain an explanation of an error returned by the *utimens*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *utimens*(2) system call.

*data*
: The original data, exactly as passed to the *utimens*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimens(pathname, data) < 0)
{
   fprintf(stderr, "%s\n", explain_utimens(pathname, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_utimens_or_die*(3) function.

### explain_errno_utimens

const char *explain_errno_utimens(int errnum, const char *pathname, const struct timespec *data);

The **explain_errno_utimens** function is used to obtain an explanation of an error returned by the *utimens*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *utimens*(2) system call.

*data*
: The original data, exactly as passed to the *utimens*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimens(pathname, data) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_utimens(err, pathname,
    data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_utimens_or_die*(3) function.

**explain_message_utimens**

void explain_message_utimens(char *message, int message_size, const char *pathname, const struct timespec *data);

The **explain_message_utimens** function is used to obtain an explanation of an error returned by the *utimens*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *utimens*(2) system call.

*data* The original data, exactly as passed to the *utimens*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimens(pathname, data) < 0)
{
    char message[3000];
    explain_message_utimens(message, sizeof(message), pathname,
    data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_utimens_or_die*(3) function.

**explain_message_errno_utimens**

void explain_message_errno_utimens(char *message, int message_size, int errnum, const char *pathname, const struct timespec *data);

The **explain_message_errno_utimens** function is used to obtain an explanation of an error returned by the *utimens*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *utimens*(2) system call.

*data* The original data, exactly as passed to the *utimens*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimens(pathname, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_utimens(message, sizeof(message), err,
   pathname, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_utimens_or_die*(3) function.

## SEE ALSO

*utimens*(2)
change file last access and modification times

*explain_utimens_or_die*(3)
change file last access and modification times and report errors

## COPYRIGHT

## NAME

explain_utimensat − explain *utimensat*(2) errors

## SYNOPSIS

#include <libexplain/utimensat.h>

const char *explain_utimensat(int fildes, const char *pathname, const struct timespec *data, int flags);
const char *explain_errno_utimensat(int errnum, int fildes, const char *pathname, const struct timespec *data, int flags);
void explain_message_utimensat(char *message, int message_size, int fildes, const char *pathname, const struct timespec *data, int flags);
void explain_message_errno_utimensat(char *message, int message_size, int errnum, int fildes, const char *pathname, const struct timespec *data, int flags);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *utimensat*(2) system call.

### explain_utimensat

const char *explain_utimensat(int fildes, const char *pathname, const struct timespec *data, int flags);

The **explain_utimensat** function is used to obtain an explanation of an error returned by the *utimensat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *utimensat*(2) system call.

*pathname* The original pathname, exactly as passed to the *utimensat*(2) system call.

*data* The original data, exactly as passed to the *utimensat*(2) system call.

*flags* The original flags, exactly as passed to the *utimensat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimensat(fildes, pathname, data, flags) < 0)
{
   fprintf(stderr, "%s\n", explain_utimensat(fildes, pathname,
   data, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_utimensat_or_die*(3) function.

### explain_errno_utimensat

const char *explain_errno_utimensat(int errnum, int fildes, const char *pathname, const struct timespec *data, int flags);

The **explain_errno_utimensat** function is used to obtain an explanation of an error returned by the *utimensat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *utimensat*(2) system call.

*pathname*
The original pathname, exactly as passed to the *utimensat*(2) system call.

*data* The original data, exactly as passed to the *utimensat*(2) system call.

*flags* The original flags, exactly as passed to the *utimensat*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimensat(fildes, pathname, data, flags) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_utimensat(err, fildes,
   pathname, data, flags));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_utimensat_or_die*(3) function.

**explain_message_utimensat**

void explain_message_utimensat(char *message, int message_size, int fildes, const char *pathname, const struct timespec *data, int flags);

The **explain_message_utimensat** function is used to obtain an explanation of an error returned by the *utimensat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *utimensat*(2) system call.

*pathname*
The original pathname, exactly as passed to the *utimensat*(2) system call.

*data* The original data, exactly as passed to the *utimensat*(2) system call.

*flags* The original flags, exactly as passed to the *utimensat*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimensat(fildes, pathname, data, flags) < 0)
{
    char message[3000];
   explain_message_utimensat(message, sizeof(message), fildes,
   pathname, data, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_utimensat_or_die*(3) function.

**explain_message_errno_utimensat**

void explain_message_errno_utimensat(char *message, int message_size, int errnum, int fildes, const char *pathname, const struct timespec *data, int flags);

The **explain_message_errno_utimensat** function is used to obtain an explanation of an error returned by the *utimensat*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *utimensat*(2) system call.

*pathname*
The original pathname, exactly as passed to the *utimensat*(2) system call.

*data* The original data, exactly as passed to the *utimensat*(2) system call.

*flags* The original flags, exactly as passed to the *utimensat*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimensat(fildes, pathname, data, flags) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_utimensat(message, sizeof(message), err,
   fildes, pathname, data, flags);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre−packaged as the *explain_utimensat_or_die*(3) function.

## SEE ALSO

*utimensat*(2)
change file timestamps with nanosecond precision

*explain_utimensat_or_die*(3)
change file timestamps with nanosecond precision and report errors

## COPYRIGHT

## NAME

explain_utimensat_or_die − change file timestamps and report errors

## SYNOPSIS

#include <libexplain/utimensat.h>

void explain_utimensat_or_die(int fildes, const char *pathname, const struct timespec *data, int flags);
int explain_utimensat_on_error(int fildes, const char *pathname, const struct timespec *data, int flags);

## DESCRIPTION

The **explain_utimensat_or_die** function is used to call the *utimensat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_utimensat*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_utimensat_on_error** function is used to call the *utimensat*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_utimensat*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *utimensat*(2) system call.

*pathname*
The pathname, exactly as to be passed to the *utimensat*(2) system call.

*data* The data, exactly as to be passed to the *utimensat*(2) system call.

*flags* The flags, exactly as to be passed to the *utimensat*(2) system call.

## RETURN VALUE

The **explain_utimensat_or_die** function only returns on success, see *utimensat*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_utimensat_on_error** function always returns the value return by the wrapped *utimensat*(2) system call.

## EXAMPLE

The **explain_utimensat_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_utimensat_or_die(fildes, pathname, data, flags);
```

## SEE ALSO

*utimensat*(2)
change file timestamps with nanosecond precision

*explain_utimensat*(3)
explain *utimensat*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_utimens_or_die − change file timestamps and report errors

## SYNOPSIS

#include <libexplain/utimens.h>

void explain_utimens_or_die(const char *pathname, const struct timespec *data);
int explain_utimens_on_error(const char *pathname, const struct timespec *data);

## DESCRIPTION

The **explain_utimens_or_die** function is used to call the *utimens*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_utimens*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_utimens_on_error** function is used to call the *utimens*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_utimens*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *utimens*(2) system call.

*data*
: The data, exactly as to be passed to the *utimens*(2) system call.

## RETURN VALUE

The **explain_utimens_or_die** function only returns on success, see *utimens*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_utimens_on_error** function always returns the value return by the wrapped *utimens*(2) system call.

## EXAMPLE

The **explain_utimens_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_utimens_or_die(pathname, data);
```

## SEE ALSO

*utimens*(2)
: change file last access and modification times

*explain_utimens*(3)
: explain *utimens*(2) errors

*exit*(2)
: terminate the calling process

## COPYRIGHT

## NAME

explain_utime_or_die − change file times and report errors

## SYNOPSIS

#include <libexplain/utime.h>

void explain_utime_or_die(const char *pathname, const struct utimbuf *times);

## DESCRIPTION

The **explain_utime_or_die** function is used to call the *utime*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_utime*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_utime_or_die(pathname, times);
```

*pathname*
: The pathname, exactly as to be passed to the *utime*(2) system call.

*times*
: The times, exactly as to be passed to the *utime*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*utime*(2) change file last access and modification times

*explain_utime*(3)
: explain *utime*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_utimes − explain *utimes*(2) errors

## SYNOPSIS

#include <libexplain/utimes.h>

const char *explain_utimes(const char *pathname, const struct timeval *data);
const char *explain_errno_utimes(int errnum, const char *pathname, const struct timeval *data);
void explain_message_utimes(char *message, int message_size, const char *pathname, const struct timeval *data);
void explain_message_errno_utimes(char *message, int message_size, int errnum, const char *pathname, const struct timeval *data);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *utimes*(2) system call.

### explain_utimes

const char *explain_utimes(const char *pathname, const struct timeval *data);

The **explain_utimes** function is used to obtain an explanation of an error returned by the *utimes*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*pathname*
: The original pathname, exactly as passed to the *utimes*(2) system call.

*data*
: The original data, exactly as passed to the *utimes*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimes(pathname, data) < 0)
{
   fprintf(stderr, "%s\n", explain_utimes(pathname, data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_utimes_or_die*(3) function.

### explain_errno_utimes

const char *explain_errno_utimes(int errnum, const char *pathname, const struct timeval *data);

The **explain_errno_utimes** function is used to obtain an explanation of an error returned by the *utimes*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
: The original pathname, exactly as passed to the *utimes*(2) system call.

*data*
: The original data, exactly as passed to the *utimes*(2) system call.

Returns:
: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any

libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimes(pathname, data) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_utimes(err, pathname,
    data));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_utimes_or_die*(3) function.

**explain_message_utimes**

void explain_message_utimes(char *message, int message_size, const char *pathname, const struct timeval *data);

The **explain_message_utimes** function is used to obtain an explanation of an error returned by the *utimes*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pathname*
The original pathname, exactly as passed to the *utimes*(2) system call.

*data* The original data, exactly as passed to the *utimes*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimes(pathname, data) < 0)
{
    char message[3000];
    explain_message_utimes(message, sizeof(message), pathname,
    data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_utimes_or_die*(3) function.

**explain_message_errno_utimes**

void explain_message_errno_utimes(char *message, int message_size, int errnum, const char *pathname, const struct timeval *data);

The **explain_message_errno_utimes** function is used to obtain an explanation of an error returned by the *utimes*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*pathname*
The original pathname, exactly as passed to the *utimes*(2) system call.

*data* The original data, exactly as passed to the *utimes*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
if (utimes(pathname, data) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_utimes(message, sizeof(message), err,
   pathname, data);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_utimes_or_die*(3) function.

## SEE ALSO

*utimes*(2)
change file last access and modification times

*explain_utimes_or_die*(3)
change file last access and modification times and report errors

## COPYRIGHT

## NAME

explain_utimes_or_die − change file access and modify times and report errors

## SYNOPSIS

#include <libexplain/utimes.h>

void explain_utimes_or_die(const char *pathname, const struct timeval *data);
int explain_utimes_on_error(const char *pathname, const struct timeval *data);

## DESCRIPTION

The **explain_utimes_or_die** function is used to call the *utimes*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_utimes*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_utimes_on_error** function is used to call the *utimes*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_utimes*(3) function, but still returns to the caller.

*pathname*
: The pathname, exactly as to be passed to the *utimes*(2) system call.

*data*
: The data, exactly as to be passed to the *utimes*(2) system call.

## RETURN VALUE

The **explain_utimes_or_die** function only returns on success, see *utimes*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_utimes_on_error** function always returns the value return by the wrapped *utimes*(2) system call.

## EXAMPLE

The **explain_utimes_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_utimes_or_die(pathname, data);
```

## SEE ALSO

*utimes*(2)
: change file last access and modification times

*explain_utimes*(3)
: explain *utimes*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_vfork − explain *vfork*(2) errors

## SYNOPSIS

#include <libexplain/vfork.h>

const char *explain_vfork(void);
const char *explain_errno_vfork(int errnum, void);
void explain_message_vfork(char *message, int message_size, void);
void explain_message_errno_vfork(char *message, int message_size, int errnum, void);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *vfork*(2) system call.

### explain_vfork

const char *explain_vfork(void);

The **explain_vfork** function is used to obtain an explanation of an error returned by the *vfork*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = vfork();
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_vfork());
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vfork_or_die*(3) function.

### explain_errno_vfork

const char *explain_errno_vfork(int errnum, void);

The **explain_errno_vfork** function is used to obtain an explanation of an error returned by the *vfork*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = vfork();
if (result < 0)
{
    int err = errno;
```

```
    fprintf(stderr, "%s\n", explain_errno_vfork(err, ));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vfork_or_die*(3) function.

**explain_message_vfork**

void explain_message_vfork(char *message, int message_size, void);

The **explain_message_vfork** function is used to obtain an explanation of an error returned by the *vfork*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = vfork();
if (result < 0)
{
    char message[3000];
    explain_message_vfork(message, sizeof(message), );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vfork_or_die*(3) function.

**explain_message_errno_vfork**

void explain_message_errno_vfork(char *message, int message_size, int errnum, void);

The **explain_message_errno_vfork** function is used to obtain an explanation of an error returned by the *vfork*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
pid_t result = vfork();
if (result < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_vfork(message, sizeof(message), err, );
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vfork_or_die*(3) function.

## SEE ALSO

*vfork*(2) create a child process and block parent

*explain_vfork_or_die*(3)
create a child process and block parent and report errors

## COPYRIGHT

## NAME

explain_vfork_or_die − create a child process and report errors

## SYNOPSIS

#include <libexplain/vfork.h>

pid_t explain_vfork_or_die(void);
pid_t explain_vfork_on_error(void);

## DESCRIPTION

The **explain_vfork_or_die** function is used to call the *vfork*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vfork*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_vfork_on_error** function is used to call the *vfork*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vfork*(3) function, but still returns to the caller.

## RETURN VALUE

The **explain_vfork_or_die** function only returns on success, see *vfork*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_vfork_on_error** function always returns the value return by the wrapped *vfork*(2) system call.

## EXAMPLE

The **explain_vfork_or_die** function is intended to be used in a fashion similar to the following example:

```
pid_t result = explain_vfork_or_die();
```

## SEE ALSO

*vfork*(2) create a child process and block parent

*explain_vfork*(3)
explain *vfork*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_vfprintf − explain *vfprintf*(3) errors

## SYNOPSIS

#include <libexplain/vfprintf.h>

const char *explain_vfprintf(FILE *fp, const char *format, va_list ap);
const char *explain_errno_vfprintf(int errnum, FILE *fp, const char *format, va_list ap);
void explain_message_vfprintf(char *message, int message_size, FILE *fp, const char *format, va_list ap);
void explain_message_errno_vfprintf(char *message, int message_size, int errnum, FILE *fp, const char *format, va_list ap);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *vfprintf*(3) system call.

### explain_vfprintf

const char *explain_vfprintf(FILE *fp, const char *format, va_list ap);

The **explain_vfprintf** function is used to obtain an explanation of an error returned by the *vfprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fp* The original fp, exactly as passed to the *vfprintf*(3) system call.

*format* The original format, exactly as passed to the *vfprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vfprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL; if (vfprintf(fp, format, ap) < 0)
{
   fprintf(stderr, "%s\n", explain_vfprintf(fp, format, ap));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vfprintf_or_die*(3) function.

### explain_errno_vfprintf

const char *explain_errno_vfprintf(int errnum, FILE *fp, const char *format, va_list ap);

The **explain_errno_vfprintf** function is used to obtain an explanation of an error returned by the *vfprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *vfprintf*(3) system call.

*format* The original format, exactly as passed to the *vfprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vfprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL; if (vfprintf(fp, format, ap) < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_vfprintf(err, fp,
   format, ap));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vfprintf_or_die*(3) function.

**explain_message_vfprintf**

void explain_message_vfprintf(char *message, int message_size, FILE *fp, const char *format, va_list ap);

The **explain_message_vfprintf** function is used to obtain an explanation of an error returned by the *vfprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fp* The original fp, exactly as passed to the *vfprintf*(3) system call.

*format* The original format, exactly as passed to the *vfprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vfprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL; if (vfprintf(fp, format, ap) < 0)
{
    char message[3000];
   explain_message_vfprintf(message, sizeof(message), fp, format,
   ap);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vfprintf_or_die*(3) function.

**explain_message_errno_vfprintf**

void explain_message_errno_vfprintf(char *message, int message_size, int errnum, FILE *fp, const char *format, va_list ap);

The **explain_message_errno_vfprintf** function is used to obtain an explanation of an error returned by the *vfprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fp* The original fp, exactly as passed to the *vfprintf*(3) system call.

*format* The original format, exactly as passed to the *vfprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vfprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL; if (vfprintf(fp, format, ap) < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_vfprintf(message, sizeof(message), err,
   fp, format, ap);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vfprintf_or_die*(3) function.

## SEE ALSO

*vfprintf*(3)
formatted output conversion

*explain_vfprintf_or_die*(3)
formatted output conversion and report errors

## COPYRIGHT

## NAME

explain_vfprintf_or_die − formatted output conversion and report errors

## SYNOPSIS

#include <libexplain/vfprintf.h>

void explain_vfprintf_or_die(FILE *fp, const char *format, va_list ap);
int explain_vfprintf_on_error(FILE *fp, const char *format, va_list ap);

## DESCRIPTION

The **explain_vfprintf_or_die** function is used to call the *vfprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vfprintf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_vfprintf_on_error** function is used to call the *vfprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vfprintf*(3) function, but still returns to the caller.

*fp* The fp, exactly as to be passed to the *vfprintf*(3) system call.

*format* The format, exactly as to be passed to the *vfprintf*(3) system call.

*ap* The ap, exactly as to be passed to the *vfprintf*(3) system call.

## RETURN VALUE

The **explain_vfprintf_or_die** function only returns on success, see *vfprintf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_vfprintf_on_error** function always returns the value return by the wrapped *vfprintf*(3) system call.

## EXAMPLE

The **explain_vfprintf_or_die** function is intended to be used in a fashion similar to the following example:

```
explain_vfprintf_or_die(fp, format, ap);
```

## SEE ALSO

*vfprintf*(3)
formatted output conversion

*explain_vfprintf*(3)
explain *vfprintf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_vprintf − explain *vprintf*(3) errors

## SYNOPSIS

#include <libexplain/vprintf.h>

const char *explain_vprintf(const char *format, va_list ap);
const char *explain_errno_vprintf(int errnum, const char *format, va_list ap);
void explain_message_vprintf(char *message, int message_size, const char *format, va_list ap);
void explain_message_errno_vprintf(char *message, int message_size, int errnum, const char *format, va_list ap);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *vprintf*(3) system call.

### explain_vprintf

const char *explain_vprintf(const char *format, va_list ap);

The **explain_vprintf** function is used to obtain an explanation of an error returned by the *vprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*format* The original format, exactly as passed to the *vprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = vprintf(format, ap);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_vprintf(format, ap));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vprintf_or_die*(3) function.

### explain_errno_vprintf

const char *explain_errno_vprintf(int errnum, const char *format, va_list ap);

The **explain_errno_vprintf** function is used to obtain an explanation of an error returned by the *vprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*format* The original format, exactly as passed to the *vprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = vprintf(format, ap);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_vprintf(err, format,
   ap));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vprintf_or_die*(3) function.

**explain_message_vprintf**

void explain_message_vprintf(char *message, int message_size, const char *format, va_list ap);

The **explain_message_vprintf** function is used to obtain an explanation of an error returned by the *vprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*format* The original format, exactly as passed to the *vprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = vprintf(format, ap);
if (result < 0)
{
    char message[3000];
   explain_message_vprintf(message, sizeof(message), format, ap);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vprintf_or_die*(3) function.

**explain_message_errno_vprintf**

void explain_message_errno_vprintf(char *message, int message_size, int errnum, const char *format, va_list ap);

The **explain_message_errno_vprintf** function is used to obtain an explanation of an error returned by the *vprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*format* The original format, exactly as passed to the *vprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = EINVAL;
int result = vprintf(format, ap);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_vprintf(message, sizeof(message), err,
   format, ap);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vprintf_or_die*(3) function.

## SEE ALSO

*vprintf*(3)
formatted output conversion

*explain_vprintf_or_die*(3)
formatted output conversion and report errors

## COPYRIGHT

## NAME

explain_vprintf_or_die − formatted output conversion and report errors

## SYNOPSIS

#include <libexplain/vprintf.h>

int explain_vprintf_or_die(const char *format, va_list ap);
int explain_vprintf_on_error(const char *format, va_list ap);

## DESCRIPTION

The **explain_vprintf_or_die** function is used to call the *vprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vprintf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_vprintf_on_error** function is used to call the *vprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vprintf*(3) function, but still returns to the caller.

*format* The format, exactly as to be passed to the *vprintf*(3) system call.

*ap* The ap, exactly as to be passed to the *vprintf*(3) system call.

## RETURN VALUE

The **explain_vprintf_or_die** function only returns on success, see *vprintf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_vprintf_on_error** function always returns the value return by the wrapped *vprintf*(3) system call.

## EXAMPLE

The **explain_vprintf_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_vprintf_or_die(format, ap);
```

## SEE ALSO

*vprintf*(3)
formatted output conversion

*explain_vprintf*(3)
explain *vprintf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_vsnprintf − explain *vsnprintf*(3) errors

## SYNOPSIS

#include <libexplain/vsnprintf.h>

const char *explain_vsnprintf(char *data, size_t data_size, const char *format, va_list ap);
const char *explain_errno_vsnprintf(int errnum, char *data, size_t data_size, const char *format, va_list ap);
void explain_message_vsnprintf(char *message, int message_size, char *data, size_t data_size, const char *format, va_list ap);
void explain_message_errno_vsnprintf(char *message, int message_size, int errnum, char *data, size_t data_size, const char *format, va_list ap);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *vsnprintf*(3) system call.

### explain_vsnprintf

const char *explain_vsnprintf(char *data, size_t data_size, const char *format, va_list ap);

The **explain_vsnprintf** function is used to obtain an explanation of an error returned by the *vsnprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *vsnprintf*(3) system call.

*data_size*
The original data_size, exactly as passed to the *vsnprintf*(3) system call.

*format* The original format, exactly as passed to the *vsnprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vsnprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = vsnprintf(data, data_size, format, ap);
if (result < 0 && errno != 0)
{
   fprintf(stderr, "%s\n", explain_vsnprintf(data, data_size,
   format, ap));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vsnprintf_or_die*(3) function.

### explain_errno_vsnprintf

const char *explain_errno_vsnprintf(int errnum, char *data, size_t data_size, const char *format, va_list ap);

The **explain_errno_vsnprintf** function is used to obtain an explanation of an error returned by the *vsnprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be

explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *vsnprintf*(3) system call.

*data_size*
The original data_size, exactly as passed to the *vsnprintf*(3) system call.

*format* The original format, exactly as passed to the *vsnprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vsnprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = vsnprintf(data, data_size, format, ap);
if (result < 0 && errno != 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_vsnprintf(err, data,
   data_size, format, ap));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vsnprintf_or_die*(3) function.

**explain_message_vsnprintf**

void explain_message_vsnprintf(char *message, int message_size, char *data, size_t data_size, const char *format, va_list ap);

The **explain_message_vsnprintf** function is used to obtain an explanation of an error returned by the *vsnprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *vsnprintf*(3) system call.

*data_size*
The original data_size, exactly as passed to the *vsnprintf*(3) system call.

*format* The original format, exactly as passed to the *vsnprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vsnprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = vsnprintf(data, data_size, format, ap);
if (result < 0 && errno != 0)
{
    char message[3000];
   explain_message_vsnprintf(message, sizeof(message), data,
   data_size, format, ap);
    fprintf(stderr, "%s\n", message);
```

```
        exit(EXIT_FAILURE);
    }
```

The above code example is available pre-packaged as the *explain_vsnprintf_or_die*(3) function.

**explain_message_errno_vsnprintf**

void explain_message_errno_vsnprintf(char *message, int message_size, int errnum, char *data, size_t data_size, const char *format, va_list ap);

The **explain_message_errno_vsnprintf** function is used to obtain an explanation of an error returned by the *vsnprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *vsnprintf*(3) system call.

*data_size*
The original data_size, exactly as passed to the *vsnprintf*(3) system call.

*format* The original format, exactly as passed to the *vsnprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vsnprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = vsnprintf(data, data_size, format, ap);
if (result < 0 && errno != 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_vsnprintf(message, sizeof(message), err,
   data, data_size, format, ap);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vsnprintf_or_die*(3) function.

## SEE ALSO

*vsnprintf*(3)
formatted output conversion

*explain_vsnprintf_or_die*(3)
formatted output conversion and report errors

## COPYRIGHT

## NAME

explain_vsnprintf_or_die − formatted output conversion and report errors

## SYNOPSIS

#include <libexplain/vsnprintf.h>

int explain_vsnprintf_or_die(char *data, size_t data_size, const char *format, va_list ap);
int explain_vsnprintf_on_error(char *data, size_t data_size, const char *format, va_list ap);

## DESCRIPTION

The **explain_vsnprintf_or_die** function is used to call the *vsnprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vsnprintf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_vsnprintf_on_error** function is used to call the *vsnprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vsnprintf*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *vsnprintf*(3) system call.

*data_size*
The data_size, exactly as to be passed to the *vsnprintf*(3) system call.

*format* The format, exactly as to be passed to the *vsnprintf*(3) system call.

*ap* The ap, exactly as to be passed to the *vsnprintf*(3) system call.

## RETURN VALUE

The **explain_vsnprintf_or_die** function only returns on success, see *vsnprintf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_vsnprintf_on_error** function always returns the value return by the wrapped *vsnprintf*(3) system call.

## EXAMPLE

The **explain_vsnprintf_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_vsnprintf_or_die(data, data_size, format, ap);
```

## SEE ALSO

*vsnprintf*(3)
formatted output conversion

*explain_vsnprintf*(3)
explain *vsnprintf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_vsprintf − explain *vsprintf*(3) errors

## SYNOPSIS

#include <libexplain/vsprintf.h>

const char *explain_vsprintf(char *data, const char *format, va_list ap);
const char *explain_errno_vsprintf(int errnum, char *data, const char *format, va_list ap);
void explain_message_vsprintf(char *message, int message_size, char *data, const char *format, va_list ap);
void explain_message_errno_vsprintf(char *message, int message_size, int errnum, char *data, const char *format, va_list ap);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *vsprintf*(3) system call.

### explain_vsprintf

const char *explain_vsprintf(char *data, const char *format, va_list ap);

The **explain_vsprintf** function is used to obtain an explanation of an error returned by the *vsprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*data* The original data, exactly as passed to the *vsprintf*(3) system call.

*format* The original format, exactly as passed to the *vsprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vsprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = vsprintf(data, format, ap);
if (result < 0 && errno != 0)
{
   fprintf(stderr, "%s\n", explain_vsprintf(data, format, ap));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vsprintf_or_die*(3) function.

### explain_errno_vsprintf

const char *explain_errno_vsprintf(int errnum, char *data, const char *format, va_list ap);

The **explain_errno_vsprintf** function is used to obtain an explanation of an error returned by the *vsprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *vsprintf*(3) system call.

*format* The original format, exactly as passed to the *vsprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vsprintf*(3) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = vsprintf(data, format, ap);
if (result < 0 && errno != 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_vsprintf(err, data,
   format, ap));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vsprintf_or_die*(3) function.

**explain_message_vsprintf**

void explain_message_vsprintf(char *message, int message_size, char *data, const char *format, va_list ap);

The **explain_message_vsprintf** function is used to obtain an explanation of an error returned by the *vsprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*data* The original data, exactly as passed to the *vsprintf*(3) system call.

*format* The original format, exactly as passed to the *vsprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vsprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = vsprintf(data, format, ap);
if (result < 0 && errno != 0)
{
    char message[3000];
   explain_message_vsprintf(message, sizeof(message), data,
   format, ap);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vsprintf_or_die*(3) function.

**explain_message_errno_vsprintf**

void explain_message_errno_vsprintf(char *message, int message_size, int errnum, char *data, const char *format, va_list ap);

The **explain_message_errno_vsprintf** function is used to obtain an explanation of an error returned by the *vsprintf*(3) system call. The least the message will contain is the value of `strerror(errno)`, but

usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*data* The original data, exactly as passed to the *vsprintf*(3) system call.

*format* The original format, exactly as passed to the *vsprintf*(3) system call.

*ap* The original ap, exactly as passed to the *vsprintf*(3) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
errno = 0;
int result = vsprintf(data, format, ap);
if (result < 0 && errno != 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_vsprintf(message, sizeof(message), err,
   data, format, ap);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_vsprintf_or_die*(3) function.

## SEE ALSO

*vsprintf*(3)
formatted output conversion

*explain_vsprintf_or_die*(3)
formatted output conversion and report errors

## COPYRIGHT

## NAME

explain_vsprintf_or_die − formatted output conversion and report errors

## SYNOPSIS

#include <libexplain/vsprintf.h>

int explain_vsprintf_or_die(char *data, const char *format, va_list ap);
int explain_vsprintf_on_error(char *data, const char *format, va_list ap);

## DESCRIPTION

The **explain_vsprintf_or_die** function is used to call the *vsprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vsprintf*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_vsprintf_on_error** function is used to call the *vsprintf*(3) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_vsprintf*(3) function, but still returns to the caller.

*data* The data, exactly as to be passed to the *vsprintf*(3) system call.

*format* The format, exactly as to be passed to the *vsprintf*(3) system call.

*ap* The ap, exactly as to be passed to the *vsprintf*(3) system call.

## RETURN VALUE

The **explain_vsprintf_or_die** function only returns on success, see *vsprintf*(3) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_vsprintf_on_error** function always returns the value return by the wrapped *vsprintf*(3) system call.

## EXAMPLE

The **explain_vsprintf_or_die** function is intended to be used in a fashion similar to the following example:

```
int result = explain_vsprintf_or_die(data, format, ap);
```

## SEE ALSO

*vsprintf*(3)
formatted output conversion

*explain_vsprintf*(3)
explain *vsprintf*(3) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_wait − explain wait(2) errors

## SYNOPSIS

#include <libexplain/wait.h>

const char *explain_wait(int *status);
const char *explain_errno_wait(int errnum, int *status);
void explain_message_wait(char *message, int message_size, int *status);
void explain_message_errno_wait(char *message, int message_size, int errnum, int *status);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *wait*(2) system call.

### explain_wait

const char *explain_wait(int *status);

The **explain_wait** function is used to obtain an explanation of an error returned by the *wait*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (wait(status) < 0)
{
    fprintf(stderr, "%s\n", explain_wait(status));
    exit(EXIT_FAILURE);
}
```

*status* The original status, exactly as passed to the *wait*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_wait

const char *explain_errno_wait(int errnum, int *status);

The **explain_errno_wait** function is used to obtain an explanation of an error returned by the *wait*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (wait(status) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_wait(err, status));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*status* The original status, exactly as passed to the *wait*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_wait**

void explain_message_wait(char *message, int message_size, int *status);

The **explain_message_wait** function may be used to obtain an explanation of an error returned by the *wait*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (wait(status) < 0)
{
    char message[3000];
    explain_message_wait(message, sizeof(message), status);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*status* The original status, exactly as passed to the *wait*(2) system call.

**explain_message_errno_wait**

void explain_message_errno_wait(char *message, int message_size, int errnum, int *status);

The **explain_message_errno_wait** function may be used to obtain an explanation of an error returned by the *wait*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (wait(status) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_wait(message, sizeof(message), err, status);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*status* The original status, exactly as passed to the *wait*(2) system call.

## SEE ALSO

*wait*(2) wait for process to change state

*explain_wait_or_die*(3)
wait for process to change state and report errors

## COPYRIGHT

## NAME

explain_wait3 − explain wait3(2) errors

## SYNOPSIS

#include <libexplain/wait3.h>

const char *explain_wait3(int *status, int options, struct rusage *rusage);
const char *explain_errno_wait3(int errnum, int *status, int options, struct rusage *rusage);
void explain_message_wait3(char *message, int message_size, int *status, int options, struct rusage *rusage);
void explain_message_errno_wait3(char *message, int message_size, int errnum, int *status, int options, struct rusage *rusage);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *wait3*(2) system call.

### explain_wait3

const char *explain_wait3(int *status, int options, struct rusage *rusage);

The **explain_wait3** function is used to obtain an explanation of an error returned by the *wait3*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int pid = wait3(status, options, rusage);
if (pid < 0)
{
    fprintf(stderr, "%s\n", explain_wait3(status, options, rusage));
    exit(EXIT_FAILURE);
}
```

*status* The original status, exactly as passed to the *wait3*(2) system call.

*options* The original options, exactly as passed to the *wait3*(2) system call.

*rusage* The original rusage, exactly as passed to the *wait3*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_wait3

const char *explain_errno_wait3(int errnum, int *status, int options, struct rusage *rusage);

The **explain_errno_wait3** function is used to obtain an explanation of an error returned by the *wait3*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int pid = wait3(status, options, rusage);
if (pid < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_wait3(err, status, options,
        rusage));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*status* The original status, exactly as passed to the *wait3*(2) system call.

*options* The original options, exactly as passed to the *wait3*(2) system call.

*rusage* The original rusage, exactly as passed to the *wait3*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_wait3**

void explain_message_wait3(char *message, int message_size, int *status, int options, struct rusage *rusage);

The **explain_message_wait3** function may be used to obtain an explanation of an error returned by the *wait3*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
int pid = wait3(status, options, rusage);
if (pid < 0)
{
    char message[3000];
    explain_message_wait3(message, sizeof(message),
        status, options, rusage);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*status* The original status, exactly as passed to the *wait3*(2) system call.

*options* The original options, exactly as passed to the *wait3*(2) system call.

*rusage* The original rusage, exactly as passed to the *wait3*(2) system call.

**explain_message_errno_wait3**

void explain_message_errno_wait3(char *message, int message_size, int errnum, int *status, int options, struct rusage *rusage);

The **explain_message_errno_wait3** function may be used to obtain an explanation of an error returned by the *wait3*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
int pid = wait3(status, options, rusage);
if (pid < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_wait3(message, sizeof(message), err,
```

```
            status, options, rusage);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*status* The original status, exactly as passed to the *wait3*(2) system call.

*options* The original options, exactly as passed to the *wait3*(2) system call.

*rusage* The original rusage, exactly as passed to the *wait3*(2) system call.

## SEE ALSO

*wait3*(2) wait for process to change state

*explain_wait3_or_die*(3)
wait for process to change state and report errors

## COPYRIGHT

## NAME

explain_wait3_or_die − wait for process to change state and report errors

## SYNOPSIS

#include <libexplain/wait3.h>

void explain_wait3_or_die(int *status, int options, struct rusage *rusage);

## DESCRIPTION

The **explain_wait3_or_die** function is used to call the *wait3*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_wait3*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
int pid = explain_wait3_or_die(status, options, rusage);
```

*status* The status, exactly as to be passed to the *wait3*(2) system call.

*options* The options, exactly as to be passed to the *wait3*(2) system call.

*rusage* The rusage, exactly as to be passed to the *wait3*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*wait3*(2) wait for process to change state

*explain_wait3*(3)
explain *wait3*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_wait4 − explain wait4(2) errors

## SYNOPSIS

#include <libexplain/wait4.h>

const char *explain_wait4(int pid, int *status, int options, struct rusage *rusage);
const char *explain_errno_wait4(int errnum, int pid, int *status, int options, struct rusage *rusage);
void explain_message_wait4(char *message, int message_size, int pid, int *status, int options, struct rusage *rusage);
void explain_message_errno_wait4(char *message, int message_size, int errnum, int pid, int *status, int options, struct rusage *rusage);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *wait4*(2) system call.

### explain_wait4

const char *explain_wait4(int pid, int *status, int options, struct rusage *rusage);

The **explain_wait4** function is used to obtain an explanation of an error returned by the *wait4*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (wait4(pid, status, options, rusage) < 0)
{
    fprintf(stderr, "%s\n", explain_wait4(pid, status, options, rusage));
    exit(EXIT_FAILURE);
}
```

*pid* The original pid, exactly as passed to the *wait4*(2) system call.

*status* The original status, exactly as passed to the *wait4*(2) system call.

*options* The original options, exactly as passed to the *wait4*(2) system call.

*rusage* The original rusage, exactly as passed to the *wait4*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_wait4

const char *explain_errno_wait4(int errnum, int pid, int *status, int options, struct rusage *rusage);

The **explain_errno_wait4** function is used to obtain an explanation of an error returned by the *wait4*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (wait4(pid, status, options, rusage) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_wait4(err,
        pid, status, options, rusage));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *wait4*(2) system call.

*status* The original status, exactly as passed to the *wait4*(2) system call.

*options* The original options, exactly as passed to the *wait4*(2) system call.

*rusage* The original rusage, exactly as passed to the *wait4*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_wait4**

void explain_message_wait4(char *message, int message_size, int pid, int *status, int options, struct rusage *rusage);

The **explain_message_wait4** function may be used to obtain an explanation of an error returned by the *wait4*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (wait4(pid, status, options, rusage) < 0)
{
    char message[3000];
    explain_message_wait4(message, sizeof(message),
        pid, status, options, rusage);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pid* The original pid, exactly as passed to the *wait4*(2) system call.

*status* The original status, exactly as passed to the *wait4*(2) system call.

*options* The original options, exactly as passed to the *wait4*(2) system call.

*rusage* The original rusage, exactly as passed to the *wait4*(2) system call.

**explain_message_errno_wait4**

void explain_message_errno_wait4(char *message, int message_size, int errnum, int pid, int *status, int options, struct rusage *rusage);

The **explain_message_errno_wait4** function may be used to obtain an explanation of an error returned by the *wait4*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (wait4(pid, status, options, rusage) < 0)
{
    int err = errno;
    char message[3000];
```

```
        explain_message_errno_wait4(message, sizeof(message), err,
            pid, status, options, rusage);
        fprintf(stderr, "%s\n", message);
        exit(EXIT_FAILURE);
    }
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *wait4*(2) system call.

*status* The original status, exactly as passed to the *wait4*(2) system call.

*options* The original options, exactly as passed to the *wait4*(2) system call.

*rusage* The original rusage, exactly as passed to the *wait4*(2) system call.

## SEE ALSO

*wait4*(2) wait for process to change state

*explain_wait4_or_die*(3)
wait for process to change state and report errors

## COPYRIGHT

## NAME

explain_wait4_or_die − wait for process to change state and report errors

## SYNOPSIS

#include <libexplain/wait4.h>

void explain_wait4_or_die(int pid, int *status, int options, struct rusage *rusage);

## DESCRIPTION

The **explain_wait4_or_die** function is used to call the *wait4*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_wait4*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_wait4_or_die(pid, status, options, rusage);
```

*pid* The pid, exactly as to be passed to the *wait4*(2) system call.

*status* The status, exactly as to be passed to the *wait4*(2) system call.

*options* The options, exactly as to be passed to the *wait4*(2) system call.

*rusage* The rusage, exactly as to be passed to the *wait4*(2) system call.

Returns: This function only returns on success, see *wait4*(2) for more information. On failure, prints an explanation and exits.

## SEE ALSO

*wait4*(2) wait for process to change state

*explain_wait4*(3)
explain *wait4*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_wait_or_die − wait for process to change state and report errors

## SYNOPSIS

#include <libexplain/wait.h>

void explain_wait_or_die(int *status);

## DESCRIPTION

The **explain_wait_or_die** function is used to call the *wait*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_wait*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_wait_or_die(status);
```

*status* The status, exactly as to be passed to the *wait*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*wait*(2) wait for process to change state

*explain_wait*(3)
explain *wait*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_waitpid − explain waitpid(2) errors

## SYNOPSIS

#include <libexplain/waitpid.h>

const char *explain_waitpid(int pid, int *status, int options);
const char *explain_errno_waitpid(int errnum, int pid, int *status, int options);
void explain_message_waitpid(char *message, int message_size, int pid, int *status, int options);
void explain_message_errno_waitpid(char *message, int message_size, int errnum, int pid, int *status, int options);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *waitpid*(2) system call.

### explain_waitpid

const char *explain_waitpid(int pid, int *status, int options);

The **explain_waitpid** function is used to obtain an explanation of an error returned by the *waitpid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (waitpid(pid, status, options) < 0)
{
    fprintf(stderr, "%s\n", explain_waitpid(pid, status, options));
    exit(EXIT_FAILURE);
}
```

*pid* The original pid, exactly as passed to the *waitpid*(2) system call.

*status* The original status, exactly as passed to the *waitpid*(2) system call.

*options* The original options, exactly as passed to the *waitpid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_waitpid

const char *explain_errno_waitpid(int errnum, int pid, int *status, int options);

The **explain_errno_waitpid** function is used to obtain an explanation of an error returned by the *waitpid*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (waitpid(pid, status, options) < 0)
{
    int err = errno;
    fprintf(stderr, "%s\n", explain_errno_waitpid(err,
        pid, status, options));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid* The original pid, exactly as passed to the *waitpid*(2) system call.

*status* The original status, exactly as passed to the *waitpid*(2) system call.

*options* The original options, exactly as passed to the *waitpid*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_waitpid**

void explain_message_waitpid(char *message, int message_size, int pid, int *status, int options);

The **explain_message_waitpid** function may be used to obtain an explanation of an error returned by the *waitpid*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

This function is intended to be used in a fashion similar to the following example:

```
if (waitpid(pid, status, options) < 0)
{
    char message[3000];
    explain_message_waitpid(message, sizeof(message), pid, status, options);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*pid* The original pid, exactly as passed to the *waitpid*(2) system call.

*status* The original status, exactly as passed to the *waitpid*(2) system call.

*options* The original options, exactly as passed to the *waitpid*(2) system call.

**explain_message_errno_waitpid**

void explain_message_errno_waitpid(char *message, int message_size, int errnum, int pid, int *status, int options);

The **explain_message_errno_waitpid** function may be used to obtain an explanation of an error returned by the *waitpid*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
if (waitpid(pid, status, options) < 0)
{
    int err = errno;
    char message[3000];
    explain_message_errno_waitpid(message, sizeof(message), err,
        pid, status, options);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
: The size in bytes of the location in which to store the returned message.

*errnum*
: The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*pid*
: The original pid, exactly as passed to the *waitpid*(2) system call.

*status*
: The original status, exactly as passed to the *waitpid*(2) system call.

*options*
: The original options, exactly as passed to the *waitpid*(2) system call.

## SEE ALSO

*waitpid*(2)
: wait for process to change state

*explain_waitpid_or_die*(3)
: wait for process to change state and report errors

## COPYRIGHT

## NAME

explain_waitpid_or_die − wait for process to change state and report errors

## SYNOPSIS

#include <libexplain/waitpid.h>

int pid = explain_waitpid_or_die(int pid, int *status, int options);

## DESCRIPTION

The **explain_waitpid_or_die** function is used to call the *waitpid*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_waitpid*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
explain_waitpid_or_die(pid, status, options);
```

*pid* The pid, exactly as to be passed to the *waitpid*(2) system call.

*status* The status, exactly as to be passed to the *waitpid*(2) system call.

*options* The options, exactly as to be passed to the *waitpid*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*waitpid*(2)
wait for process to change state

*explain_waitpid*(3)
explain *waitpid*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_write − explain write(2) errors

## SYNOPSIS

#include <libexplain/write.h>
const char *explain_write(int fildes, const void *data, long data_size);
const char *explain_errno_write(int errnum, int fildes, const void *data, long data_size);
void explain_message_write(char *message, int message_size, int fildes, const void *data, long data_size);
void explain_message_errno_write(char *message, int message_size, int errnum, int fildes, const void *data, long data_size);

## DESCRIPTION

These functions may be used to obtain explanations for *write*(2) errors .

### explain_write

const char *explain_write(int fildes, const void *data, long data_size);

The explain_write function may be used to obtain a human readable explanation of what went wrong in a *write*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The error number will be picked up from the *errno* global variable.

This function is intended to be used in a fashion similar to the following example:

```
sszie_t n = write(fd, data, data_size);
if (n < 0)
{
    fprintf(stderr, '%s0, explain_read(fd, data, data_size));
    exit(EXIT_FAILURE);
}
```

*fildes* The original fildes, exactly as passed to the *write*(2) system call.

*data* The original data, exactly as passed to the *write*(2) system call.

*data_size*
The original data_size, exactly as passed to the *write*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

### explain_errno_write

const char *explain_errno_write(int errnum, int fildes, const void *data, long data_size);

The explain_errno_write function may be used to obtain a human readable explanation of what went wrong in a *write*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
sszie_t n = write(fd, data, data_size);
if (n < 0)
{
    int err = errno;
    fprintf(stderr, '%s0, explain_errno_read(errnum, fd, data,
        data_size));
    exit(EXIT_FAILURE);
}
```

*errnum* The error value to be decoded, usually obtain from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The orginal fildes, exactly as passed to the *write*(2) system call.

*data* The original data, exactly as passed to the *write*(2) system call.

*data_size*
The original data_size, exactly as passed to the *write*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**explain_message_write**

void explain_message_write(char *message, int message_size, int fildes, const void *data, long data_size);

The explain_message_write function may be used to obtain a human readable explanation of what went wrong in a *write*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The error number will be picked up from the *errno* global variable.

This function is intended to be used in a fashion similar to the following example:

```
sszie_t n = write(fd, data, data_size);
if (n < 0)
{
    char message[3000];
    explain_message_read(message, sizeof(message), fd, data,
        data_size));
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *write*(2) system call.

*data* The original data, exactly as passed to the *write*(2) system call.

*data_size*
The original data_size, exactly as passed to the *write*(2) system call.

**Note:** Given a suitably thread safe buffer, this function is thread safe.

**explain_message_errno_write**

void explain_message_errno_write(char * message, int message_size, int errnum, int fildes, const void *data, long data_size);

The explain_message_errno_write function may be used to obtain a human readable explanation of what went wrong in a *write*(2) system call. The least the message will contain is the value of `strerror(errnum)`, but usually it will do much better, and indicate the underlying cause in more detail.

This function is intended to be used in a fashion similar to the following example:

```
sszie_t n = write(fd, data, data_size);
if (n < 0)
```

```
{
    int err = errno;
    char message[3000];
    explain_message_errno_read(message, sizeof(message), errno,
        fd, data, data_size));
    fprintf(stderr, '%s0, message);
    exit(EXIT_FAILURE);
}
```

*message* The location in which to store the returned message. Because a message return buffer has been supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtain from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *write*(2) system call.

*data* The original data, exactly as passed to the *write*(2) system call.

*data_size*
The original data_size, exactly as passed to the *write*(2) system call.

**Note:** Given a suitably thread safe buffer, this function is thread safe.

## COPYRIGHT



## AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>

## NAME

explain_write_or_die − write to a file descriptor and report errors

## SYNOPSIS

#include <libexplain/write.h>

void explain_write_or_die(int fildes, const void *data, long data_size);

## DESCRIPTION

The **explain_write_or_die** function is used to call the *write*(2) system call. On failure an explanation will be printed to *stderr*, obtained from *explain_write*(3), and then the process terminates by calling `exit(EXIT_FAILURE)`.

This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = explain_write_or_die(fildes, data, data_size);
```

*fildes* The fildes, exactly as to be passed to the *write*(2) system call.

*data* The data, exactly as to be passed to the *write*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *write*(2) system call.

Returns: This function only returns on success. On failure, prints an explanation and exits.

## SEE ALSO

*write*(2) write to a file descriptor

*explain_write*(3)
explain *write*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT

## NAME

explain_writev − explain writev(2) errors

## SYNOPSIS

#include <libexplain/writev.h>

const char *explain_writev(int fildes, const struct iovec *data, int data_size);
const char *explain_errno_writev(int errnum, int fildes, const struct iovec *data, int data_size);
void explain_message_writev(char *message, int message_size, int fildes, const struct iovec *data, int data_size);
void explain_message_errno_writev(char *message, int message_size, int errnum, int fildes, const struct iovec *data, int data_size);

## DESCRIPTION

These functions may be used to obtain explanations for errors returned by the *writev*(2) system call.

### explain_writev

const char *explain_writev(int fildes, const struct iovec *data, int data_size);

The **explain_writev** function is used to obtain an explanation of an error returned by the *writev*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*fildes* The original fildes, exactly as passed to the *writev*(2) system call.

*data* The original data, exactly as passed to the *writev*(2) system call.

*data_size*
The original data_size, exactly as passed to the *writev*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = writev(fildes, data, data_size);
if (result < 0)
{
   fprintf(stderr, "%s\n", explain_writev(fildes, data,
   data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_writev_or_die*(3) function.

### explain_errno_writev

const char *explain_errno_writev(int errnum, int fildes, const struct iovec *data, int data_size);

The **explain_errno_writev** function is used to obtain an explanation of an error returned by the *writev*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *writev*(2) system call.

*data* The original data, exactly as passed to the *writev*(2) system call.

*data_size*
The original data_size, exactly as passed to the *writev*(2) system call.

Returns: The message explaining the error. This message buffer is shared by all libexplain functions which do not supply a buffer in their argument list. This will be overwritten by the next call to any libexplain function which shares this buffer, including other threads.

**Note:** This function is **not** thread safe, because it shares a return buffer across all threads, and many other functions in this library.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = writev(fildes, data, data_size);
if (result < 0)
{
    int err = errno;
   fprintf(stderr, "%s\n", explain_errno_writev(err, fildes,
   data, data_size));
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_writev_or_die*(3) function.

**explain_message_writev**

void explain_message_writev(char *message, int message_size, int fildes, const struct iovec *data, int data_size);

The **explain_message_writev** function is used to obtain an explanation of an error returned by the *writev*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

The *errno* global variable will be used to obtain the error value to be decoded.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*fildes* The original fildes, exactly as passed to the *writev*(2) system call.

*data* The original data, exactly as passed to the *writev*(2) system call.

*data_size*
The original data_size, exactly as passed to the *writev*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = writev(fildes, data, data_size);
if (result < 0)
{
    char message[3000];
   explain_message_writev(message, sizeof(message), fildes, data,
   data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_writev_or_die*(3) function.

**explain_message_errno_writev**

void explain_message_errno_writev(char *message, int message_size, int errnum, int fildes, const struct iovec *data, int data_size);

The **explain_message_errno_writev** function is used to obtain an explanation of an error returned by the

*writev*(2) system call. The least the message will contain is the value of `strerror(errno)`, but usually it will do much better, and indicate the underlying cause in more detail.

*message* The location in which to store the returned message. If a suitable message return buffer is supplied, this function is thread safe.

*message_size*
The size in bytes of the location in which to store the returned message.

*errnum* The error value to be decoded, usually obtained from the *errno* global variable just before this function is called. This is necessary if you need to call **any** code between the system call to be explained and this function, because many libc functions will alter the value of *errno*.

*fildes* The original fildes, exactly as passed to the *writev*(2) system call.

*data* The original data, exactly as passed to the *writev*(2) system call.

*data_size*
The original data_size, exactly as passed to the *writev*(2) system call.

**Example:** This function is intended to be used in a fashion similar to the following example:

```
ssize_t result = writev(fildes, data, data_size);
if (result < 0)
{
    int err = errno;
    char message[3000];
   explain_message_errno_writev(message, sizeof(message), err,
   fildes, data, data_size);
    fprintf(stderr, "%s\n", message);
    exit(EXIT_FAILURE);
}
```

The above code example is available pre-packaged as the *explain_writev_or_die*(3) function.

## SEE ALSO

*writev*(2)
write data from multiple buffers

*explain_writev_or_die*(3)
write data from multiple buffers and report errors

## COPYRIGHT

## NAME

explain_writev_or_die − write data from multiple buffers and report errors

## SYNOPSIS

#include <libexplain/writev.h>

ssize_t explain_writev_or_die(int fildes, const struct iovec *data, int data_size);
ssize_t explain_writev_on_error(int fildes, const struct iovec *data, int data_size);

## DESCRIPTION

The **explain_writev_or_die** function is used to call the *writev*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_writev*(3) function, and then the process terminates by calling `exit(EXIT_FAILURE)`.

The **explain_writev_on_error** function is used to call the *writev*(2) system call. On failure an explanation will be printed to *stderr*, obtained from the *explain_writev*(3) function, but still returns to the caller.

*fildes* The fildes, exactly as to be passed to the *writev*(2) system call.

*data* The data, exactly as to be passed to the *writev*(2) system call.

*data_size*
The data_size, exactly as to be passed to the *writev*(2) system call.

## RETURN VALUE

The **explain_writev_or_die** function only returns on success, see *writev*(2) for more information. On failure, prints an explanation and exits, it does not return.

The **explain_writev_on_error** function always returns the value return by the wrapped *writev*(2) system call.

## EXAMPLE

The **explain_writev_or_die** function is intended to be used in a fashion similar to the following example:

```
ssize_t result = explain_writev_or_die(fildes, data, data_size);
```

## SEE ALSO

*writev*(2)
write data from multiple buffers

*explain_writev*(3)
explain *writev*(2) errors

*exit*(2) terminate the calling process

## COPYRIGHT