The GNU C Library Manual

Release 2.24

Many

August 20, 2016

Contents:

CHAPTER 1

Introduction

The C language provides no built-in facilities for performing such common operations as input/output, memory management, string manipulation, and the like. Instead, these facilities are defined in a standard library, which you compile and link with your programs.

The GNU C Library, described in this document, defines all of the library functions that are specified by the ISO C standard, as well as additional features specific to POSIX and other derivatives of the Unix operating system, and extensions specific to GNU systems. The purpose of this manual is to tell you how to use the facilities of the GNU C Library. We have mentioned which features belong to which standards to help you identify things that are potentially non-portable to other systems. But the emphasis in this manual is not on strict portability.

1.1 Getting Started

This manual is written with the assumption that you are at least somewhat familiar with the C programming language and basic programming concepts. Specifically, familiarity with ISO standard C (see Section 1.2.1 [ISO C], page 2), rather than "traditional" pre-ISO C dialects, is assumed.

The GNU C Library includes several header files, each of which provides definitions and declarations for a group of related facilities; this information is used by the C compiler when processing your program. For example, the header file stdio.h declares facilities for performing input and output, and the header file string.h declares string processing utilities. The organization of this manual generally follows the same division as the header files.

If you are reading this manual for the first time, you should read all of the introductory material and skim the remaining chapters. There are a lot of functions in the GNU C Library and it's not realistic to expect that you will be able to remember exactly how to use each and every one of them. It's more important to become generally familiar with the kinds of facilities that the library provides, so that when you are writing your programs you can recognize when to make use of library functions, and where in this manual you can find more specific information about them.

1.2 Standards and Portability

This section discusses the various standards and other sources that the GNU C Library is based upon. These sources include the ISO C and POSIX standards, and the System V and Berkeley Unix implementations.

The primary focus of this manual is to tell you how to make effective use of the GNU C Library facilities. But if you are concerned about making your programs compatible with these standards, or portable to operating systems other than GNU, this can affect how you use the library. This section gives you an overview of these standards, so that you will know what they are when they are mentioned in other parts of the manual.

See Appendix B [Summary of Library Facilities], page 897, for an alphabetical list of the functions and other symbols provided by the library. This list also states which standards each function or symbol comes from.

1.2.1 ISO C

The GNU C Library is compatible with the C standard adopted by the American National Standards Institute (ANSI): American National Standard X3.159-1989—"ANSI C" and later by the International Standardization Organization (ISO): ISO/IEC 9899:1990, "Programming languages—C". We here refer to the standard as ISO C since this is the more general standard in respect of ratification. The header files and library facilities that make up the GNU C Library are a superset of those specified by the ISO C standard.

If you are concerned about strict adherence to the ISO C standard, you should use the '-ansi' option when you compile your programs with the GNU C compiler. This tells the compiler to define only ISO standard features from the library header files, unless you explicitly ask for additional features. See Section 1.3.4 [Feature Test Macros], page 15, for information on how to do this

Being able to restrict the library to include only ISO C features is important because ISO C puts limitations on what names can be defined by the library implementation, and the GNU extensions don't fit these limitations. See Section 1.3.3 [Reserved Names], page 14, for more information about these restrictions.

This manual does not attempt to give you complete details on the differences between ISO C and older dialects. It gives advice on how to write programs to work portably under multiple C dialects, but does not aim for completeness.

1.2.2 POSIX (The Portable Operating System Interface)

The library facilities specified by the POSIX standards are a superset of those required by ISO C; POSIX specifies additional features for ISO C functions, as well as specifying new additional functions. In general, the additional requirements and functionality defined by the POSIX standards are aimed at providing lower-level support for a particular kind of operating system environment, rather than general programming language support which can run in many diverse operating system environments.

The GNU C Library implements all of the functions specified in ISO/IEC 9945-1:1996, the POSIX System Application Program Interface, commonly referred to as POSIX.1. The primary extensions to the ISO C facilities specified by this standard include file system interface primitives (see Chapter 14 [File System Interface], page 379), device-specific terminal control functions (see Chapter 17 [Low-Level Terminal Interface], page 479), and process control functions (see Chapter 26 [Processes], page 752).

Some facilities from ISO/IEC 9945-2:1993, the POSIX Shell and Utilities standard (POSIX.2) are also implemented in the GNU C Library. These include utilities for dealing with regular expressions and other pattern matching facilities (see Chapter 10 [Pattern Matching], page 223).

POSIX Safety Concepts

This manual documents various safety properties of GNU C Library functions, in lines that follow their prototypes and look like:

Preliminary: | MT-Safe | AS-Safe | AC-Safe

The properties are assessed according to the criteria set forth in the POSIX standard for such safety contexts as Thread-, Async-Signal- and Async-Cancel- -Safety. Intuitive definitions of these properties, attempting to capture the meaning of the standard definitions, follow.

- MT-Safe or Thread-Safe functions are safe to call in the presence of other threads. MT, in MT-Safe, stands for Multi Thread. Being MT-Safe does not imply a function is atomic, nor that it uses any of the memory synchronization mechanisms POSIX exposes to users. It is even possible that calling MT-Safe functions in sequence does not yield an MT-Safe combination. For example, having a thread call two MT-Safe functions one right after the other does not guarantee behavior equivalent to atomic execution of a combination of both functions, since concurrent calls in other threads may interfere in a destructive way. Whole-program optimizations that could inline functions across library interfaces may expose unsafe reordering, and so performing inlining across the GNU C Library interface is not recommended. The documented MT-Safety status is not guaranteed under whole-program optimization. However, functions defined in user-visible headers are designed to be safe for inlining.

- AS-Safe or Async-Signal-Safe functions are safe to call from asynchronous signal handlers. AS, in AS-Safe, stands for Asynchronous Signal. Many functions that are AS-Safe may set errno, or modify the floating-point environment, because their doing so does not make them unsuitable for use in signal handlers. However, programs could misbehave should asynchronous signal handlers modify this thread-local state, and the signal handling machinery cannot be counted on to preserve it. Therefore, signal handlers that call functions that may set errno or modify the floating-point environment must save their original values, and restore them before returning.
- AC-Safe or Async-Cancel-Safe functions are safe to call when asynchronous cancellation is enabled. AC in AC-Safe stands for Asynchronous Cancellation. The POSIX standard defines only three functions to be AC-Safe, namely **pthread_** cancel, pthread_setcancelstate, and pthread_setcanceltype. At present the GNU C Library provides no guarantees beyond these three functions, but does document which functions are presently AC-Safe. This documentation is provided for use by the GNU C Library developers. Just like signal handlers, cancellation cleanup routines must configure the floating point environment they require. The routines cannot assume a floating point environment, particularly when asynchronous cancellation is enabled. If the configuration of the floating point environment cannot be performed atomically then it is also possible that the environment encountered is internally inconsistent.
- MT-Unsafe, AS-Unsafe, AC-Unsafe functions are not safe to call within the safety contexts described above. Calling them within such contexts invokes undefined behavior. Functions not explicitly documented as safe in a safety context should be regarded as Unsafe.
- Preliminary safety properties are documented, indicating these properties may not be counted on in future releases of the GNU C Library.

Such preliminary properties are the result of an assessment of the properties of our current implementation, rather than of what is mandated and permitted by current and future standards.

Although we strive to abide by the standards, in some cases our implementation is safe even when the standard does not demand safety, and in other cases our implementation does not meet the standard safety requirements. The latter are most likely bugs; the former, when marked as Preliminary, should not be counted on: future standards may require changes that are not compatible with the additional safety properties afforded by the current implementation

Furthermore, the POSIX standard does not offer a detailed definition of safety. We assume that, by "safe to call", POSIX means that, as long as the program does not invoke undefined behavior, the "safe to call" function behaves as specified, and does not cause other functions to deviate from their specified behavior. We have chosen to use its loose definitions of safety, not because they are the best definitions to use, but because choosing them harmonizes this manual with POSIX

Over time, we envision evolving the preliminary safety notes into stable commitments, as stable as those of our interfaces. As we do, we will remove the Preliminary keyword from safety notes. As long as the keyword remains, however, they are not to be regarded as a promise of future behavior.

Other keywords that appear in safety notes are defined in subsequent sections.

Unsafe Features

Functions that are unsafe to call in certain contexts are annotated with keywords that document their features that make them unsafe to call. AS-Unsafe features in this section indicate the functions are never safe to call when asynchronous signals are enabled. AC-Unsafe features indicate they are never safe to call when asynchronous cancellation is enabled. There are no MT-Unsafe marks in this section.

- lock Functions marked with lock as an AS-Unsafe feature may be interrupted by a signal while holding a non-recursive lock. If the signal handler calls another such function that takes the same lock, the result is a deadlock. Functions annotated with lock as an AC-Unsafe feature may, if cancelled asynchronously, fail to release a lock that would have been released if their execution had not been interrupted by asynchronous thread cancellation. Once a lock is left taken, attempts to take that lock will block indefinitely.

- corrupt Functions marked with corrupt as an AS-Unsafe feature may corrupt data structures and misbehave when they interrupt, or are interrupted by, another such function. Unlike functions marked with lock, these take recursive locks to avoid MT-Safety problems, but this is not enough to stop a signal handler from observing a partiallyupdated data structure. Further corruption may arise from the interrupted function's failure to notice updates made by signal handlers. Functions marked with corrupt as an AC-Unsafe feature may leave data structures in a corrupt, partially updated state. Subsequent uses of the data structure may misbehave.
- heap Functions marked with heap may call heap memory management functions from the malloc/free family of functions and are only as safe as those functions. This note is thus equivalent to: | AS-Unsafe lock | AC-Unsafe lock fd mem |
- dlopen Functions marked with dlopen use the dynamic loader to load shared libraries into the current execution image. This involves opening files, mapping them into memory, allocating additional memory, resolving symbols, applying relocations and more, all of this while holding internal dynamic loader locks. The locks are enough for these functions to be AS- and AC-Unsafe, but other issues may arise. At present this is a placeholder for all potential safety issues raised by dlopen.
- plugin Functions annotated with plugin may run code from plugins that may be external to the GNU C Library. Such plugin functions are assumed to be MT-Safe, AS-Unsafe and AC-Unsafe. Examples of such plugins are stack unwinding libraries, name service switch (NSS) and character set conversion (iconv) back-ends. Although the plugins mentioned as examples are all brought in by means of dlopen, the plugin keyword does not imply any direct involvement of the dynamic loader or the libdl interfaces, those are covered by dlopen. For example, if one function loads a module and finds the addresses of some of its functions, while another just calls those already-resolved functions, the former will be marked with dlopen, whereas the latter will get the plugin. When a single function takes all of these actions, then it gets both marks.
- i18n Functions marked with i18n may call internationalization functions of the gettext family and will be only as safe as those functions. This note is thus equivalent to: | MT-Safe env | AS-Unsafe corrupt heap dlopen | AC-Unsafe corrupt |
- timer Functions marked with timer use the alarm function or similar to set a time-out for a system call or a long-running operation. In a multi-threaded program, there is a risk that the time-out signal will be delivered to a different thread, thus failing to interrupt the intended thread. Besides being MT-Unsafe, such functions are always AS-Unsafe, because calling them in signal handlers may interfere with timers set in the interrupted code, and AC-Unsafe, because there is no safe way to guarantee an earlier timer will be reset in case of asynchronous cancellation.

Conditionally Safe Features

For some features that make functions unsafe to call in certain contexts, there are known ways to avoid the safety problem other than refraining from calling the function altogether. The keywords that follow refer to such features, and each of their definitions indicate how the whole program needs to be constrained in order to remove the safety problem indicated by the keyword. Only when all the reasons that make a function unsafe are observed and addressed, by applying the documented constraints, does the function become safe to call in a context.

- init Functions marked with init as an MT-Unsafe feature perform MT-Unsafe initialization when they are first called. Calling such a function at least once in single-threaded mode removes this specific cause for the function to be regarded as MT-Unsafe. If no other cause for that remains, the function can then be safely called after other threads are started. Functions marked with init as an AS- or AC-Unsafe feature use the internal **libc_** once machinery or similar to initialize internal data structures. If a signal handler interrupts such an initializer, and calls any function that also performs libc_once initialization, it will deadlock if the thread library has been loaded. Furthermore, if an initializer is partially complete before it is canceled or interrupted by a signal whose handler requires the same initialization, some or all of the initialization may be performed more than once, leaking resources or even resulting in corrupt internal data. Applications that need to call functions marked with init as an AS- or AC-Unsafe feature should ensure the initialization is performed before configuring signal handlers or enabling cancellation, so that the AS- and AC-Safety issues related with libc_once do not arise.

- race Functions annotated with race as an MT-Safety issue operate on objects in ways that may cause data races or similar forms of destructive interference out of concurrent execution. In some cases, the objects are passed to the functions by users; in others, they are used by the functions to return values to users; in others, they are not even exposed to users. We consider access to objects passed as (indirect) arguments to functions to be data race free. The assurance of data race free objects is the caller's responsibility. We will not mark a function as MT-Unsafe or AS-Unsafe if it misbehaves when users fail to take the measures required by POSIX to avoid data races when dealing with such objects. As a general rule, if a function is documented as reading from an object passed (by reference) to it, or modifying it, users ought to use memory synchronization primitives to avoid data races just as they would should they perform the accesses themselves rather than by calling the library function. FILE streams are the exception to the general rule, in that POSIX mandates the library to guard against data races in many functions that manipulate objects of this specific opaque type. We regard this as a convenience provided to users, rather than as a general requirement whose expectations should extend to other types. In order to remind users that guarding certain arguments is their responsibility, we will annotate functions that take objects of certain types as arguments. We draw the line for objects passed by users as follows: objects whose types are exposed to users, and that users are expected to access directly, such as memory buffers, strings, and various user-visible struct types, do not give reason for functions to be annotated with race. It would be noisy and redundant with the general requirement, and not many would be surprised by the library's lack of internal guards when accessing objects that can be accessed directly by users. As for objects that are opaque or opaque-like, in that they are to be manipulated only by passing them to library functions (e.g., FILE, DIR, obstack, iconv_t), there might be additional expectations as to internal coordination of access by the library. We will annotate, with race followed by a colon and the argument name, functions that take such objects but that do not take care of synchronizing access to them by default. For example, FILE stream unlocked functions will be annotated, but those that perform implicit locking on FILE streams by default will not, even though the implicit locking may be disabled on a per-stream basis. In either case, we will not regard as MT-Unsafe functions that may access user-supplied objects in unsafe ways should users fail to ensure the accesses are well defined. The notion prevails that users are expected to safeguard against data races any user-supplied objects that the library accesses on their behalf. This user responsibility does not apply, however, to objects controlled by the library itself, such as internal objects and static buffers used to return values from certain calls. When the library doesn't guard them against concurrent uses, these cases are regarded as MT-Unsafe and AS-Unsafe (although the race mark under AS-Unsafe will be omitted as redundant with the one under MT-Unsafe). As in the case of userexposed objects, the mark may be followed by a colon and an identifier. The identifier groups all functions that operate on a certain unguarded object; users may avoid the MT-Safety issues related with unguarded concurrent access to such internal objects by creating a non-recursive mutex related with the identifier, and always holding the mutex when calling any function marked as racy on that identifier, as they would have to should the identifier be an object under user control. The non-recursive mutex avoids the MT-Safety issue, but it trades one AS-Safety issue for another, so use in asynchronous signals remains undefined. When the identifier relates to a static buffer used to hold return values, the mutex must be held for as long as the buffer remains in use by the caller. Many functions that return pointers to static buffers offer reentrant variants that store return values in caller-supplied buffers instead. In some cases, such as tmpname, the variant is chosen not by calling an alternate entry point, but by passing a non-NULL pointer to the buffer in which the returned values are to be stored. These variants are generally preferable in multi-threaded programs, although some of them are not MT-Safe because of other internal buffers, also documented with race notes.
- const Functions marked with const as an MT-Safety issue non-atomically modify internal objects that are better regarded as constant, because a substantial portion of the GNU C Library accesses them without synchronization. Unlike race, that causes both readers and writers of internal objects to be regarded as MT-Unsafe and AS-Unsafe, this mark is applied to writers only. Writers remain equally MT- and AS-Unsafe to call, but the then-mandatory constness of objects they modify enables readers to be regarded as MT-Safe and AS-Safe (as long as no other reasons for them to be unsafe remain), since the lack of synchronization is not a problem when the objects are effectively constant. The identifier that follows the const mark will appear by itself as a safety note in readers. Programs that wish to work around this safety issue, so as to call writers, may use a non-recursve rwlock associated with the identifier, and guard all calls to functions marked with const followed

by the identifier with a write lock, and all calls to functions marked with the identifier by itself with a read lock. The non-recursive locking removes the MT-Safety problem, but it trades one AS-Safety problem for another, so use in asynchronous signals remains undefined.

- sig Functions marked with sig as a MT-Safety issue (that implies an identical AS-Safety issue, omitted for brevity) may temporarily install a signal handler for internal purposes, which may interfere with other uses of the signal, identified after a colon. This safety problem can be worked around by ensuring that no other uses of the signal will take place for the duration of the call. Holding a non-recursive mutex while calling all functions that use the same temporary signal; blocking that signal before the call and resetting its handler afterwards is recommended. There is no safe way to guarantee the original signal handler is restored in case of asynchronous cancellation, therefore so-marked functions are also AC-Unsafe. Besides the measures recommended to work around the MT- and AS-Safety problem, in order to avert the cancellation problem, disabling asynchronous cancellation and installing a cleanup handler to restore the signal to the desired state and to release the mutex are recommended.
- term Functions marked with term as an MT-Safety issue may change the terminal settings in the recommended way, namely: call tcgetattr, modify some flags, and then call tcsetattr; this creates a window in which changes made by other threads are lost. Thus, functions marked with term are MT-Unsafe. The same window enables changes made by asynchronous signals to be lost. These functions are also AS-Unsafe, but the corresponding mark is omitted as redundant. It is thus advisable for applications using the terminal to avoid concurrent and reentrant interactions with it, by not using it in signal handlers or blocking signals that might use it, and holding a lock while calling these functions and interacting with the terminal. This lock should also be used for mutual exclusion with functions marked with race:tcattr(fd), where fd is a file descriptor for the controlling terminal. The caller may use a single mutex for simplicity, or use one mutex per terminal, even if referenced by different file descriptors. Functions marked with term as an AC-Safety issue are supposed to restore terminal settings to their original state, after temporarily changing them, but they may fail to do so if cancelled. Besides the measures recommended to work around the MT- and AS-Safety problem, in order to avert the cancellation problem, disabling asynchronous cancellation and installing a cleanup handler to restore the terminal settings to the original state and to release the mutex are recommended.

Other Safety Remarks

Additional keywords may be attached to functions, indicating features that do not make a function unsafe to call, but that may need to be taken into account in certain classes of programs:

- locale Functions annotated with locale as an MT-Safety issue read from the locale object without any form of synchronization. Functions annotated with locale called concurrently with locale changes may behave in ways that do not correspond to any of the locales active during their execution, but an unpredictable mix thereof. We do not mark these functions as MT- or AS-Unsafe, however, because functions that modify the locale object are marked with const:locale and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the locale can be considered effectively constant in these contexts, which makes the former safe.
- env Functions marked with env as an MT-Safety issue access the environment with getenv or similar, without any guards to ensure safety in the presence of concurrent modifications. We do not mark these functions as MT- or AS-Unsafe, however, because functions that modify the environment are all marked with const:env and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the environment can be considered effectively constant in these contexts, which makes the former safe.
- hostid The function marked with hostid as an MT-Safety issue reads from the system-wide data structures that hold the “host ID” of the machine. These data structures cannot generally be modified atomically. Since it is expected that the “host ID” will not normally change, the function that reads from it (gethostid) is regarded as safe, whereas the function that modifies it (sethostid) is marked with const:hostid, indicating it may require special care if it is to be called. In this specific case, the special care amounts to system-wide (not merely intra-process) coordination.

- sigintr Functions marked with sigintr as an MT-Safety issue access the _sigintr internal data structure without any guards to ensure safety in the presence of concurrent modifications. We do not mark these functions as MT- or AS-Unsafe, however, because functions that modify the this data structure are all marked with const:sigintr and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the data structure can be considered effectively constant in these contexts, which makes the former saf
- fd Functions annotated with fd as an AC-Safety issue may leak file descriptors if asynchronous thread cancellation interrupts their execution. Functions that allocate or deallocate file descriptors will generally be marked as such. Even if they attempted to protect the file descriptor allocation and deallocation with cleanup regions, allocating a new descriptor and storing its number where the cleanup region could release it cannot be performed as a single atomic operation. Similarly, releasing the descriptor and taking it out of the data structure normally responsible for releasing it cannot be performed atomically. There will always be a window in which the descriptor cannot be released because it was not stored in the cleanup handler argument yet, or it was already taken out before releasing it. It cannot be taken out after release: an open descriptor could mean either that the descriptor still has to be closed, or that it already did so but the descriptor was reallocated by another thread or signal handler. Such leaks could be internally avoided, with some performance penalty, by temporarily disabling asynchronous thread cancellation. However, since callers of allocation or deallocation functions would have to do this themselves, to avoid the same sort of leak in their own layer, it makes more sense for the library to assume they are taking care of it than to impose a performance penalty that is redundant when the problem is solved in upper layers, and insufficient when it is not. This remark by itself does not cause a function to be regarded as AC-Unsafe. However, cumulative effects of such leaks may pose a problem for some programs. If this is the case, suspending asynchronous cancellation for the duration of calls to such functions is recommended.
- mem Functions annotated with mem as an AC-Safety issue may leak memory if asynchronous thread cancellation interrupts their execution. The problem is similar to that of file descriptors: there is no atomic interface to allocate memory and store its address in the argument to a cleanup handler, or to release it and remove its address from that argument, without at least temporarily disabling asynchronous cancellation, which these functions do not do. This remark does not by itself cause a function to be regarded as generally AC-Unsafe. However, cumulative effects of such leaks may be severe enough for some programs that disabling asynchronous cancellation for the duration of calls to such functions may be required.
- cwd Functions marked with cwd as an MT-Safety issue may temporarily change the current working directory during their execution, which may cause relative pathnames to be resolved in unexpected ways in other threads or within asynchronous signal or cancellation handlers. This is not enough of a reason to mark so-marked functions as MT- or AS-Unsafe, but when this behavior is optional (e.g., nftw with FTW_CHDIR), avoiding the option may be a good alternative to using full pathnames or file descriptor-relative (e.g. openat) system calls.
- !posix This remark, as an MT-, AS- or AC-Safety note to a function, indicates the safety status of the function is known to differ from the specified status in the POSIX standard. For example, POSIX does not require a function to be Safe, but our implementation is, or vice-versa. For the time being, the absence of this remark does not imply the safety properties we documented are identical to those mandated by POSIX for the corresponding functions.
- :identifier Annotations may sometimes be followed by identifiers, intended to group several functions that e.g. access the data structures in an unsafe way, as in race and const, or to provide more specific information, such as naming a signal in a function marked with sig. It is envisioned that it may be applied to lock and corrupt as well in the future. In most cases, the identifier will name a set of functions, but it may name global objects or function arguments, or identifiable properties or logical components associated with them, with a notation such as e.g. :buf(arg) to denote a buffer associated with the argument arg, or :tcattr(fd) to denote the terminal attributes of a file descriptor fd. The most common use for identifiers is to provide logical groups of functions and arguments that need to be protected by the same synchronization primitive in order to ensure safe operation in a given context.
- /condition Some safety annotations may be conditional, in that they only apply if a boolean expression involving arguments, global variables or even the underlying kernel evaluates to true. Such conditions as /hurd or /!linux!bsd indicate the preceding marker only applies when the underlying kernel is the HURD, or when it is

neither Linux nor a BSD kernel, respectively. /!ps and /one_per_line indicate the preceding marker only applies when argument ps is NULL, or global variable one per line is nonzero. When all marks that render a function unsafe are adorned with such conditions, and none of the named conditions hold, then the function can be regarded as safe.

1.2.3 Berkeley Unix

The GNU C Library defines facilities from some versions of Unix which are not formally standardized, specifically from the 4.2 BSD, 4.3 BSD, and 4.4 BSD Unix systems (also known as Berkeley Unix) and from SunOS (a popular 4.2 BSD derivative that includes some Unix System V functionality). These systems support most of the ISO C and POSIX facilities, and 4.4 BSD and newer releases of SunOS in fact support them all.

The BSD facilities include symbolic links (see Section 14.5 [Symbolic Links], page 395), the select function (see Section 13.8 [Waiting for Input or Output], page 344), the BSD signal functions (see Section 24.10 [BSD Signal Handling], page 706), and sockets (see Chapter 16 [Sockets], page 431).

1.2.4 SVID (The System V Interface Description)

The System V Interface Description (SVID) is a document describing the AT&T Unix System V operating system. It is to some extent a superset of the POSIX standard (see Section 1.2.2 [POSIX (The Portable Operating System Interface)], page 2).

The GNU C Library defines most of the facilities required by the SVID that are not also required by the ISO C or POSIX standards, for compatibility with System V Unix and other Unix systems (such as SunOS) which include these facilities. However, many of the more obscure and less generally useful facilities required by the SVID are not included. (In fact, Unix System V itself does not provide them all.)

The supported facilities from System V include the methods for inter-process communication and shared memory, the hsearch and drand48 families of functions, fmtmsg and several of the mathematical functions.

1.2.5 XPG (The X/Open Portability Guide)

The X/Open Portability Guide, published by the X/Open Company, Ltd., is a more general standard than POSIX. X/Open owns the Unix copyright and the XPG specifies the requirements for systems which are intended to be a Unix system.

The GNU C Library complies to the X/Open Portability Guide, Issue 4.2, with all extensions common to XSI (X/Open System Interface) compliant systems and also all X/Open UNIX extensions.

The additions on top of POSIX are mainly derived from functionality available in System V and BSD systems. Some of the really bad mistakes in System V systems were corrected, though. Since fulfilling the XPG standard with the Unix extensions is a precondition for getting the Unix brand chances are good that the functionality is available on commercial systems.

1.3 Using the Library

This section describes some of the practical issues involved in using the GNU C Library.

1.3.1 Header Files

Libraries for use by C programs really consist of two parts: header files that define types and macros and declare variables and functions; and the actual library or archive that contains the definitions of the variables and functions.

(Recall that in C, a declaration merely provides information that a function or variable exists and gives its type. For a function declaration, information about the types of its arguments might be provided as well. The purpose of declarations is to allow the compiler to correctly process references to the declared variables and functions. A definition, on the other hand, actually allocates storage for a variable or says what a function does.)

In order to use the facilities in the GNU C Library, you should be sure that your program source files include the appropriate header files. This is so that the compiler has declarations of these facilities available and can correctly process references to them. Once your program has been compiled, the linker resolves these references to the actual definitions provided in the archive file.

Header files are included into a program source file by the '#include' preprocessor directive. The C language supports two forms of this directive; the firs

```
#include "header"
```

is typically used to include a header file header that you write yourself; this would contain definitions and declarations describing the interfaces between the different parts of your particular application. By contrast,

```
#include <file.h>
```

is typically used to include a header file file.h that contains definitions and declarations for a standard library. This file would normally be installed in a standard place by your system administrator. You should use this second form for the C library header files.

Typically, '#include' directives are placed at the top of the C source file, before any other code. If you begin your source files with some comments explaining what the code in the file does (a good idea), put the '#include' directives immediately afterwards, following the feature test macro definition (see Section 1.3.4 [Feature Test Macros], page 15).

For more information about the use of header files and '#include' directives, see Section "Header Files" in *The GNU C Preprocessor Manual*.

The GNU C Library provides several header files, each of which contains the type and macro definitions and variable and function declarations for a group of related facilities. This means that your programs may need to include several header files, depending on exactly which facilities you are using.

Some library header files include other library header files automatically. However, as a matter of programming style, you should not rely on this; it is better to explicitly include all the header files required for the library facilities you are using. The GNU C Library header files have been written in such a way that it doesn't matter if a header file is accidentally included more than once; including a header file a second time has no effect. Likewise, if your program needs to include multiple header files, the order in which they are included doesn't matter.

Compatibility Note: Inclusion of standard header files in any order and any number of times works in any ISO C implementation. However, this has traditionally not been the case in many older C implementations.

Strictly speaking, you don't have to include a header file to use a function it declares; you could declare the function explicitly yourself, according to the specifications in this manual. But it is usually better to include the header file because it may define types and macros that are not otherwise available and because it may define more efficient macro replacements for some functions. It is also a sure way to have the correct declaration.

1.3.2 Macro Definitions of Functions

If we describe something as a function in this manual, it may have a macro definition as well. This normally has no effect on how your program runs—the macro definition does the same thing as the function would. In particular, macro equivalents for library functions evaluate arguments exactly once, in the same way that a function call would. The main reason for these macro definitions is that sometimes they can produce an inline expansion that is considerably faster than an actual function call.

Taking the address of a library function works even if it is also defined as a macro. This is because, in this context, the name of the function isn't followed by the left parenthesis that is syntactically necessary to recognize a macro call.

You might occasionally want to avoid using the macro definition of a function—perhaps to make your program easier to debug. There are two ways you can do this:

- You can avoid a macro definition in a specific use by enclosing the name of the function in parentheses. This works because the name of the function don't appear in a syntactic context where it is recognizable as a macro call.
- You can suppress any macro definition for a whole source file by using the '#undef' preprocessor directive, unless otherwise stated explicitly in the description of that facility. For example, suppose the header file stdlib.h declares a function named abs with

```
    extern int abs (int);
```

and also provides a macro definition for abs. Then, in:

```
#include <stdlib.h>
int f (int *i) { return abs (++*i); }
```

the reference to abs might refer to either a macro or a function. On the other hand, in each of the following examples the reference is to a function and not a macro.

```
#include <stdlib.h>
int g (int *i) { return (abs) (++*i); }
#undef abs
int h (int *i) { return abs (++*i); }
```

Since macro definitions that double for a function behave in exactly the same way as the actual function version, there is usually no need for any of these methods. In fact, removing macro definitions usually just makes your program slower.

1.3.3 Reserved Names

The names of all library types, macros, variables and functions that come from the ISO C standard are reserved unconditionally; your program may not redefine these names. All other library names are reserved if your program explicitly includes the header file that defines or declares them. There are several reasons for these restrictions:

- Other people reading your code could get very confused if you were using a function named exit to do something completely different from what the standard exit function does, for example. Preventing this situation helps to make your programs easier to understand and contributes to modularity and maintainability.
- It avoids the possibility of a user accidentally redefining a library function that is called by other library functions. If redefinition were allowed, those other functions would not work properly.
- It allows the compiler to do whatever special optimizations it pleases on calls to these functions, without the possibility that they may have been redefined by the user. Some library facilities, such as those for dealing with variadic arguments (see Section A.2 [Variadic Functions], page 882) and non-local exits (see Chapter 23 [Non-Local Exits], page 655), actually require a considerable amount of cooperation on the part of the C compiler, and with respect to the implementation, it might be easier for the compiler to treat these as built-in parts of the language.

In addition to the names documented in this manual, reserved names include all external identifiers (global functions and variables) that begin with an underscore ('_') and all identifiers regardless of use that begin with either two underscores or an underscore followed by a capital letter are reserved names. This is so that the library and header files can define functions, variables, and macros for internal purposes without risk of conflict with names in user programs.

Some additional classes of identifier names are reserved for future extensions to the C language or the POSIX.1 environment. While using these names for your own purposes right now might not cause a problem, they do raise the possibility of conflict with future versions of the C or POSIX standards, so you should avoid these names.

- Names beginning with a capital 'E' followed a digit or uppercase letter may be used for additional error code names. See Chapter 2 [Error Reporting], page 22.
- Names that begin with either 'is' or 'to' followed by a lowercase letter may be used for additional character testing and conversion functions. See Chapter 4 [Character Handling], page 76.
- Names that begin with '**LC_**' followed by an uppercase letter may be used for additional macros specifying locale attributes. See Chapter 7 [Locales and Internationalization], page 169.
- Names of all existing mathematics functions (see Chapter 19 [Mathematics], page 514) suffixed with 'f' or 'l' are reserved for corresponding functions that operate on float
- Names that begin with 'SIG' followed by an uppercase letter are reserved for additional signal names. See Section 24.2 [Standard Signals], page 666.and long double arguments, respectively.
- Names that begin with '**SIG_**' followed by an uppercase letter are reserved for additional

  signal actions. See Section 24.3.1 [Basic Signal Handling], page 675.
- Names beginning with 'str', 'mem', or 'wcs' followed by a lowercase letter are reserved for additional string and array functions. See Chapter 5 [String and Array Utilities], page 86.
- Names that end with '_t' are reserved for additional type names.

In addition, some individual header files reserve names beyond those that they actually define. You only need to worry about these restrictions if your program includes that particular header file.

- The header file dirent.h reserves names prefixed with *d_*.
- The header file fcntl.h reserves names prefixed with *l_*, '**F_**', '**O_**', and '**S_**'.
- The header file grp.h reserves names prefixed with *gr_*.
- The header file limits.h reserves names suffixed with *_MAX*.
- The header file pwd.h reserves names prefixed with *pw_*.
- The header file signal.h reserves names prefixed with '**sa_**' and '**SA_**'.
- The header file sys/stat.h reserves names prefixed with '**st_**' and '**S_**'.
- The header file sys/times.h reserves names prefixed with '**tms_**'.
- The header file termios.h reserves names prefixed with '**c_**', 'V', 'I', 'O', and 'TC'; and names prefixed with 'B' followed by a digit.

1.3.4 Feature Test Macros

The exact set of features available when you compile a source file is controlled by which feature test macros you define.

If you compile your programs using 'gcc -ansi', you get only the ISO C library features, unless you explicitly request additional features by defining one or more of the feature macros. See Section "GNU CC Command Options" in The GNU CC Manual, for more information about GCC options.

You should define these macros by using '#define' preprocessor directives at the top of your source code files. These directives must come before any #include of a system header file. It is best to make them the very first thing in the file, preceded only by comments. You could also use the '-D' option to GCC, but it's better if you make the source files indicate their own meaning in a self-contained way.

This system exists to allow the library to conform to multiple standards. Although the different standards are often described as supersets of each other, they are usually incompatible because larger standards require functions with names that smaller ones reserve to the user program. This is not mere pedantry — it has been a problem in practice. For instance, some non-GNU programs define functions named getline that have nothing to do with this library's getline. They would not be compilable if all features were enabled indiscriminately.

This should not be used to verify that a program conforms to a limited standard. It is insufficient for this purpose, as it will not protect you from including header files outside the standard, or relying on semantics undefined within the standard.

_POSIX_SOURCE If you define this macro, then the functionality from the POSIX.1 standard (IEEE Standard 1003.1) is available, as well as all of the ISO C facilities. The state of _POSIX_SOURCE is irrelevant if you define the macro _POSIX_C_SOURCE to a positive integer.

_POSIX_C_SOURCE Define this macro to a positive integer to control which POSIX functionality is made available. The greater the value of this macro, the more functionality is made available. If you define this macro to a value greater than or equal to 1, then the functionality from the 1990 edition of the POSIX.1 standard (IEEE Standard 1003.1-1990) is made available. If you define this macro to a value greater than or equal to 2, then the functionality from the 1992 edition of the POSIX.2 standard (IEEE Standard 1003.2-1992) is made available. If you define this macro to a value greater than or equal to 199309L, then the functionality from the 1993 edition of the POSIX.1b standard (IEEE Standard 1003.1b-1993) is made available. Greater values for _POSIX_C_SOURCE will enable future extensions. The POSIX standards process will define these values as necessary, and the GNU C Library should support them some time after they become standardized. The 1996 edition of POSIX.1 (ISO/IEC 9945-1: 1996) states that if you define _POSIX_C_SOURCE to a value greater than or equal to 199506L, then the functionality from the 1996 edition is made available.

_XOPEN_SOURCE Macro

_XOPEN_SOURCE_EXTENDED If you define this macro, functionality described in the X/Open Portability Guide is included. This is a superset of the POSIX.1 and POSIX.2 functionality and in fact _POSIX_SOURCE and _POSIX_C_SOURCE are automatically defined.

As the unification of all Unices, functionality only available in BSD and SVID is also included. If the macro _XOPEN_SOURCE_EXTENDED is also defined, even more functionality is available. The extra functions will make all functions available which are necessary for the X/Open Unix brand. If the macro _XOPEN_SOURCE has the value 500 this includes all functionality described so far plus some new definitions from the Single Unix Specification, version 2.

_LARGEFILE_SOURCE If this macro is defined some extra functions are available which rectify a few shortcomings in all previous standards. Specifically, the functions fseeko and ftello are available. Without these functions the difference between the ISO C interface (fseek, ftell) and the low-level POSIX interface (lseek) would lead to problems. This macro was introduced as part of the Large File Support extension (LFS).

_LARGEFILE64_SOURCE If you define this macro an additional set of functions is made available which enables 32 bit systems to use files of sizes beyond the usual limit of 2GB. This interface is not available if the system does not support files that large. On systems where the natural file size limit is greater than 2GB (i.e., on 64 bit systems) the new functions are identical to the replaced functions. The new functionality is made available by a new set of types and functions which replace the existing ones. The names of these new objects contain 64 to indicate the intention, e.g., off_t vs. off64_t and fseeko vs. fseeko64. This macro was introduced as part of the Large File Support extension (LFS). It is a transition interface for the period when 64 bit offsets are not generally used (see _FILE_OFFSET_BITS).

_FILE_OFFSET_BITS This macro determines which file system interface shall be used, one replacing the other. Whereas _LARGEFILE64_SOURCE makes the 64 bit interface available as an additional interface, _FILE_OFFSET_BITS allows the 64 bit interface to replace the old interface. If _FILE_OFFSET_BITS is undefined, or if it is defined to the value 32, nothing changes. The 32 bit interface is used and types like off_t have a size of 32 bits on 32 bit systems. If the macro is defined to the value 64, the large file interface replaces the old interface. I.e., the functions are not made available under different names (as they are with _LARGEFILE64_SOURCE). Instead the old function names now reference the new functions, e.g., a call to fseeko now indeed calls fseeko64. This macro should only be selected if the system provides mechanisms for handling large files. On 64 bit systems this macro has no effect since the *64 functions are identical to the normal functions. This macro was introduced as part of the Large File Support extension (LFS).

_ISOC99_SOURCE Until the revised ISO C standard is widely adopted the new features are not automatically enabled. The GNU C Library nevertheless has a complete implementation of the new standard and to enable the new features the macro _ISOC99_SOURCE should be defined.

_GNU_SOURCE If you define this macro, everything is included: ISO C89, ISO C99, POSIX.1, POSIX.2, BSD, SVID, X/Open, LFS, and GNU extensions. In the cases where POSIX.1 conflicts with BSD, the POSIX definitions take precedence.

_DEFAULT_SOURCE If you define this macro, most features are included apart from X/Open, LFS and GNU extensions: the effect is to enable features from the 2008 edition of POSIX, as well as certain BSD and SVID features without a separate feature test macro to control them. Defining this macro, on its own and without using compiler options such as -ansi or -std=c99, has the same effect as not defining any feature test macros; defining it together with other feature test macros, or when options such as -ansi are used, enables those features even when the other options would otherwise cause them to be disabled.

_REENTRANT Macro

_THREAD_SAFE If you define one of these macros, reentrant versions of several functions get declared. Some of the functions are specified in POSIX.1c but many others are only available on a few other systems or are unique to the GNU C Library. The problem is the delay in the standardization of the thread safe C library interface. Unlike on some other systems, no special version of the C library must be used for linking. There is only one version but while compiling this it must have been specified to compile as thread safe.

We recommend you use _GNU_SOURCE in new programs. If you don't specify the '-ansi' option to GCC, or other conformance options such as -std=c99, and don't define any of these macros explicitly, the effect is the same as defining _DEFAULT_SOURCE to 1.

When you define a feature test macro to request a larger class of features, it is harmless to define in addition a feature test macro for a subset of those features. For example, if you define _POSIX_C_SOURCE, then defining _POSIX_SOURCE as well has no effect. Likewise, if you define _GNU_SOURCE, then defining either _POSIX_SOURCE or _POSIX_C_SOURCE as well has no effect.

1.4 Roadmap to the Manual

Here is an overview of the contents of the remaining chapters of this manual.

- Chapter 2 [Error Reporting], page 22, describes how errors detected by the library are reported.
- Chapter 3 [Virtual Memory Allocation And Paging], page 39, describes the GNU C Library's facilities for managing and using virtual and real memory, including dynamic allocation of virtual memory. If you do not know in advance how much memory your program needs, you can allocate it dynamically instead, and manipulate it via pointers.
- Chapter 4 [Character Handling], page 76, contains information about character classification functions (such as isspace) and functions for performing case conversion.
- Chapter 5 [String and Array Utilities], page 86, has descriptions of functions for manipulating strings (null-terminated character arrays) and general byte arrays, including operations such as copying and comparison.
- Chapter 6 [Character Set Handling], page 127, contains information about manipulating characters and strings using character sets larger than will fit in the usual char data type.

- Chapter 7 [Locales and Internationalization], page 169, describes how selecting a particular country or language affects the behavior of the library. For example, the locale affects collation sequences for strings and how monetary values are formatted.
- Chapter 9 [Searching and Sorting], page 213, contains information about functions for searching and sorting arrays. You can use these functions on any kind of array by providing an appropriate comparison function.
- Chapter 10 [Pattern Matching], page 223, presents functions for matching regular expressions and shell file name patterns, and for expanding words as the shell does.
- Chapter 11 [Input/Output Overview], page 245, gives an overall look at the input and output facilities in the library, and contains information about basic concepts such as file names.
- Chapter 12 [Input/Output on Streams], page 250, describes I/O operations involving streams (or FILE * objects). These are the normal C library functions from stdio.h.
- Chapter 13 [Low-Level Input/Output], page 325, contains information about I/O operations on file descriptors. File descriptors are a lower-level mechanism specific to the Unix family of operating systems.
- Chapter 14 [File System Interface], page 379, has descriptions of operations on entire files, such as functions for deleting and renaming them and for creating new directories. This chapter also contains information about how you can access the attributes of a file, such as its owner and file protection modes.
- Chapter 15 [Pipes and FIFOs], page 426, contains information about simple interprocess communication mechanisms. Pipes allow communication between two related processes (such as between a parent and child), while FIFOs allow communication between processes sharing a common file system on the same machine.
- Chapter 16 [Sockets], page 431, describes a more complicated interprocess communication mechanism that allows processes running on different machines to communicate over a network. This chapter also contains information about Internet host addressing and how to use the system network databases.
- Chapter 17 [Low-Level Terminal Interface], page 479, describes how you can change the attributes of a terminal device. If you want to disable echo of characters typed by the user, for example, read this chapter.
- Chapter 19 [Mathematics], page 514, contains information about the math library functions. These include things like random-number generators and remainder functions on integers as well as the usual trigonometric and exponential functions on floating-point numbers.
- Chapter 20 [Low-Level Arithmetic Functions], page 562, describes functions for simple arithmetic, analysis of floating-point values, and reading numbers from strings.
- Chapter 21 [Date and Time], page 598, describes functions for measuring both calendar time and CPU time, as well as functions for setting alarms and timers.
- Chapter 23 [Non-Local Exits], page 655, contains descriptions of the setjmp and longjmp functions. These functions provide a facility for goto-like jumps which can jump from one function to another.
- Chapter 24 [Signal Handling], page 664, tells you all about signals—what they are, how to establish a handler that is called when a particular kind of signal is delivered, and how to prevent signals from arriving during critical sections of your program.
- Chapter 25 [The Basic Program/System Interface], page 708, tells how your programs can access their command-line arguments and environment variables.
- Chapter 26 [Processes], page 752, contains information about how to start new processes and run programs.
- Chapter 28 [Job Control], page 765, describes functions for manipulating process groups and the controlling terminal. This material is probably only of interest if you are writing a shell or other program which handles job control specially.
- Chapter 29 [System Databases and Name Service Switch], page 784, describes the services which are available for looking up names in the system databases, how to determine which service is used for which database, and how these services are implemented so that contributors can design their own services.

- Section 30.13 [User Database], page 813, and Section 30.14 [Group Database], page 817, tell you how to access the system user and group databases.
- Chapter 31 [System Management], page 824, describes functions for controlling and getting information about the hardware and software configuration your program is executing under.
- Chapter 32 [System Configuration Parameters], page 841, tells you how you can get information about various operating system limits. Most of these parameters are provided for compatibility with POSIX.
- Appendix A [C Language Facilities in the Library], page 881, contains information about library support for standard parts of the C language, including things like the sizeof operator and the symbolic constant NULL, how to write functions accepting variable numbers of arguments, and constants describing the ranges and other properties of the numerical types. There is also a simple debugging mechanism which allows you to put assertions in your code, and have diagnostic messages printed if the tests fail.
- Appendix B [Summary of Library Facilities], page 897, gives a summary of all the functions, variables, and macros in the library, with complete data types and function prototypes, and says what standard or system each is derived from.
- Appendix C [Installing the GNU C Library], page 1000, explains how to build and install the GNU C Library on your system, and how to report any bugs you might find.
- Appendix D [Library Maintenance], page 1008, explains how to add new functions or port the library to a new system.

If you already know the name of the facility you are interested in, you can look it up in Appendix B [Summary of Library Facilities], page 897. This gives you a summary of its syntax and a pointer to where you can find a more detailed description. This appendix is particularly useful if you just want to verify the order and type of arguments to a function, for example. It also tells you what standard or system each function, variable, or macro is derived from.

CHAPTER 2

Error Reporting

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

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

2.1 Checking for Errors

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

`volatile int errno` [Variable]

> The variable `errno` contains the system error number. You can change the value of `errno`.
>
> Since `errno` is declared `volatile`, it might be changed asynchronously by a signal handler; see Section 24.4 [Defining Signal Handlers], page 681. However, a properly written signal handler saves and restores the value of `errno`, so you generally do not need to worry about this possibility except when writing signal handlers.
>
> The initial value of `errno` at program startup is zero. Many library functions are guaranteed to set it to certain nonzero values when they encounter certain kinds of errors. These error conditions are listed for each function. These functions do not change `errno` when they succeed; thus, the value of `errno` after a successful call is not necessarily zero, and you should not use `errno` to determine whether a call failed. The proper way to do that is documented for each function. If the call failed, you can examine `errno`.
>
> Many library functions can set `errno` to a nonzero value as a result of calling other library functions which might fail. You should assume that any library function might alter `errno` when the function returns an error.
>
> **Portability Note:** ISO C specifies `errno` as a "modifiable lvalue" rather than as a variable, permitting it to be implemented as a macro. For example, its expansion might involve a function call, like `*__errno_location ()`. In fact, that is what it is on GNU/Linux and GNU/Hurd systems. The GNU C Library, on each system, does whatever is right for the particular system.
>
> There are a few library functions, like sqrt and atan, that return a perfectly legitimate value in case of an error, but also set `errno`. For these functions, if you want to check to see whether an error occurred, the recommended method is to set `errno` to zero before calling the function, and then check its value afterward.

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

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

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

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

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

2.2 Error Codes

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

Macro:*int* **EPERM**

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

`int ENOENT`

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

`int ESRCH`

No process matches the specified process ID.

`int EINTR`

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

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

`int EIO`

Input/output error; usually used for physical read or write errors.

`int ENXIO`

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

`int E2BIG`

Argument list too long; used when the arguments passed to a new program being executed with one of the exec functions (see Executing a File) occupy too much memory space. This condition never arises on GNU/Hurd systems.

`int ENOEXEC`

Invalid executable file format. This condition is detected by the `exec` functions; see Executing a File.

`int EBADF`

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

`int ECHILD`

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

`int EDEADLK`

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

`int ENOMEM`

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

`int EACCES`

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

`int EFAULT`

Bad address; an invalid pointer was detected. On GNU/Hurd systems, this error never happens; you get a signal instead.

`int ENOTBLK`

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

`int EBUSY`

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

`int EEXIST`

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

`int EXDEV`

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

`int ENODEV`

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

`int ENOTDIR`

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

`int EISDIR`

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

`int EINVAL`

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

`int EMFILE`

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

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

`int ENFILE`

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

`int ENOTTY`

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

`int ETXTBSY`

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

`int EFBIG`

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

`int ENOSPC`

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

`int ESPIPE`

Invalid seek operation (such as on a pipe).

`int EROFS`

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

`int EMLINK`

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

`int EPIPE`

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

`int EDOM`

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

`int ERANGE`

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

`int EAGAIN`

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

This error can happen in a few different situations:

- An operation that would block was attempted on an object that has non-blocking mode selected. Trying the same operation again will block until some external condition makes it possible to read, write, or connect (whatever the operation). You can use select to find out when the operation will be possible; see Waiting for I/O.

  Portability Note: In many older Unix systems, this condition was indicated by `EWOULDBLOCK`, which was a distinct error code different from `EAGAIN`. To make your program portable, you should check for both codes and treat them the same.

- A temporary resource shortage made an operation impossible. `fork` can return this `error`. It indicates that the shortage is expected to pass, so your program can try the call again later and it may succeed. It is probably a good idea to delay for a few seconds before trying it again, to allow time for other processes to release scarce resources. Such shortages are usually fairly serious and affect the whole system, so usually an interactive program should report the error to the user and return to its command loop.

`int EWOULDBLOCK`

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

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

`int EINPROGRESS`

An operation that cannot complete immediately was initiated on an object that has non-blocking mode selected. Some functions that must always block (such as `connect`; see Connecting) never return `EAGAIN`. Instead, they return `EINPROGRESS` to indicate that the operation has begun and will take some time. Attempts to manipulate the object before the call completes return `EALREADY`. You can use the `select` function to find out when the pending operation has completed; see Waiting for I/O.

`int EALREADY`

An operation is already in progress on an object that has non-blocking mode selected.

`int ENOTSOCK`

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

`int EMSGSIZE`

The size of a message sent on a socket was larger than the supported maximum size.

`int EPROTOTYPE`

The socket type does not support the requested communications protocol.

`int ENOPROTOOPT`

You specified a socket option that doesn't make sense for the particular protocol being used by the socket. See Socket Options.

`int EPROTONOSUPPORT`

The socket domain does not support the requested communications protocol (perhaps because the requested protocol is completely invalid). See Creating a Socket.

`int ESOCKTNOSUPPORT`

The socket type is not supported.

`int EOPNOTSUPP`

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

`int EPFNOSUPPORT`

The socket communications protocol family you requested is not supported.

`int EAFNOSUPPORT`

The address family specified for a socket is not supported; it is inconsistent with the protocol being used on the socket. See Sockets.

`int EADDRINUSE`

The requested socket address is already in use. See Socket Addresses.

`int EADDRNOTAVAIL`

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

`int ENETDOWN`

A socket operation failed because the network was down.

`int ENETUNREACH`

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

`int ENETRESET`

A network connection was reset because the remote host crashed.

`int ECONNABORTED`

A network connection was aborted locally.

`int ECONNRESET`

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

`int ENOBUFS`

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

`int EISCONN`

You tried to connect a socket that is already connected. See Connecting.

`int ENOTCONN`

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

`int EDESTADDRREQ`

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

`int ESHUTDOWN`

The socket has already been shut down.

`int ETOOMANYREFS`

???

`int ETIMEDOUT`

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

`int ECONNREFUSED`

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

`int ELOOP`

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

`int ENAMETOOLONG`

Filename too long (longer than `PATH_MAX`; see Limits for Files) or host name too long (in `gethostname` or `sethostname`; see Host Identification).

`int EHOSTDOWN`

The remote host for a requested network connection is down.

`int EHOSTUNREACH`

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

`int ENOTEMPTY`

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

`int EPROCLIM`

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

`int EUSERS`

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

`int EDQUOT`

The user's disk quota was exceeded.

`int ESTALE`

Stale file handle. This indicates an internal confusion in the file system which is due to file system rearrangements on the server host for NFS file systems or corruption in other file systems. Repairing this condition usually requires unmounting, possibly repairing and remounting the file system.

`int EREMOTE`

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

`int EBADRPC`

???

`int ERPCMISMATCH`

???

`int EPROGUNAVAIL`

???

`int EPROGMISMATCH`

???

`int EPROCUNAVAIL`

???

`int ENOLCK`

No locks available. This is used by the file locking facilities; see File Locks. This error is never generated by GNU/Hurd systems, but it can result from an operation to an NFS server running another operating system.

`int EFTYPE`

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

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

`int EAUTH`

???

`int ENEEDAUTH`

???

`int ENOSYS`

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

`int ENOTSUP`

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

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

`int EILSEQ`

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

`int EBACKGROUND`

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

`int EDIED`

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

`int ED`

The experienced user will know what is wrong.

`int EGREGIOUS`

You did **what**?

`int EIEIO`

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

`int EGRATUITOUS`

This error code has no purpose.

`int EBADMSG`

`int EIDRM`

`int EMULTIHOP`

`int ENODATA`

`int ENOLINK`

`int ENOMSG`

`int ENOSR`

`int ENOSTR`

`int EOVERFLOW`

`int EPROTO`

`int ETIME`

`int ECANCELED`

Operation canceled; an asynchronous operation was canceled before it completed. See Asynchronous I/O. When you call `aio_cancel`, the normal result is for the operations affected to complete with this error; see Cancel AIO Operations.

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

`int ERESTART`

`int ECHRNG`

`int EL2NSYNC`

`int EL3HLT`

`int EL3RST`

`int ELNRNG`

`int EUNATCH`

`int ENOCSI`

`int EL2HLT`

`int EBADE`

`int EBADR`

`int EXFULL`

`int ENOANO`

`int EBADRQC`

`int EBADSLT`

`int EDEADLOCK`

`int EBFONT`

`int ENONET`

`int ENOPKG`

`int EADV`

`int ESRMNT`

`int ECOMM`

`int EDOTDOT`

`int ENOTUNIQ`

`int EBADFD`

`int EREMCHG`

`int ELIBACC`

`int ELIBBAD`

`int ELIBSCN`

`int ELIBMAX`

`int ELIBEXEC`

`int ESTRPIPE`

`int EUCLEAN`

`int ENOTNAM`

`int ENAVAIL`

`int EISNAM`

`int EREMOTEIO`

`int ENOMEDIUM`

`int EMEDIUMTYPE`

`int ENOKEY`

`int EKEYEXPIRED`

`int EKEYREVOKED`

`int EKEYREJECTED`

`int EOWNERDEAD`

`int ENOTRECOVERABLE`

`int ERFKILL`

`int EHWPOISON`

2.3 Error Messages

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

char * strerror (int errnum) Preliminary: | MT-Unsafe race:strerror | AS-Unsafe heap i18n | AC-Unsafe mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The strerror function maps the error code (see Section 2.1 [Checking for Errors], page 22) specified by the errnum argument to a descriptive error message string. The return value is a pointer to this string.

The value errnum normally comes from the variable errno.

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

The function strerror is declared in string.h.

char * strerror_r (int errnum, char *buf, size t n) Preliminary: | MT-Safe | AS-Unsafe i18n | AC-Unsafe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

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

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

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

The function strerror_r is a GNU extension and it is declared in string.h.

void perror (const char *message)

Preliminary: | MT-Safe race:stderr | AS-Unsafe corrupt i18n heap lock | AC-Unsafe corrupt lock mem fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. This function prints an error message to the stream stderr; see Section 12.2 [Standard Streams], page 250. The orientation of stderr is not changed. If you call perror with a message that is either a null pointer or an empty string, perror just prints the error message corresponding to errno, adding a trailing new- line. If you supply a non-null message argument, then perror prefixes its output with this string. It adds a colon and a space character to separate the message from the error string corresponding to errno. The function perror is declared in stdio.h.

strerror and perror produce the exact same message for any given error code; the

precise text varies from system to system. With the GNU C Library, the messages are fairly short; there are no multi-line messages or embedded newlines. Each error message begins with a capital letter and does not include any terminating punctuation.

Many programs that don't read input from the terminal are designed to exit if any

system call fails. By convention, the error message from such a program should start with the program's name, sans directories. You can find that name in the variable **program_** invocation_short_name; the full file name is stored the variable **program_invocation_** name.

char * program_invocation_name This variable's value is the name that was used to invoke the program running in the current process. It is the same as argv[0]. Note that this is not necessarily a useful file name; often it contains no directory names. See Section 25.1 [Program Arguments], page 708. This variable is a GNU extension and is declared in errno.h.

char * program_invocation_short_name

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

The library initialization code sets up both of these variables before calling main. **Portability Note:** If you want your program to work with non-GNU libraries, you must

save the value of argv[0] in main, and then strip off the directory names yourself. We added these extensions to make it possible to write self-contained error-reporting subroutines that require no explicit cooperation from main.

Here is an example showing how to handle failure to open a file correctly. The function

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

implementing ISO C. But often the text perror generates is not what is wanted and there is no way to extend or change what perror does. The GNU coding standard, for instance, requires error messages to be preceded by the program name and programs which read some input files should provide information about the input file name and the line number in case an error is encountered while reading the file. For these occasions there are two functions available which are widely used throughout the GNU project. These functions are declared in error.h.

void error (int status, int errnum, const char *format, . . .) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap i18n | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The error function can be used to report general problems during program execution. The format argument is a format string just like those given to the printf family of functions. The arguments required for the format can follow the format parameter. Just like perror, error also can report an error code in textual form. But unlike perror the error value is explicitly passed to the function in the errnum parameter. This eliminates the problem mentioned above that the error reporting function must be called immediately after the function causing the error since otherwise errno might have a different value. error prints first the program name. If the application defined a global variable error_print_progname and points it to a function this function will be called to print the program name. Otherwise the string from the global variable **program_** name is used. The program name is followed by a colon and a space which in turn is followed by the output produced by the format string. If the errnum parameter is non-zero the format string output is followed by a colon and a space, followed by the error message for the error code errnum. In any case is the output terminated with a newline. The output is directed to the stderr stream. If the stderr wasn't oriented before the call it will be narrow-oriented afterwards. The function will return unless the status parameter has a non-zero value. In this case the function will call exit with the status value for its parameter and therefore never return. If error returns, the global variable error_message_count is incremented by one to keep track of the number of errors reported.

void error_at_line (int status, int errnum, const char *fname,

unsigned int lineno, const char *format, . . .)

Preliminary: | MT-Unsafe race:error at line/error one per line locale | AS-Unsafe corrupt heap i18n | AC-Unsafe corrupt/error one per line | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The error_at_line function is very similar to the error function. The only dif- ferences are the additional parameters fname and lineno. The handling of the other parameters is identical to that of error except that between the program name and the string generated by the format string additional text is inserted. Directly following the program name a colon, followed by the file name pointed to by fname, another colon, and the value of lineno is printed. This additional output of course is meant to be used to locate an error in an input file (like a programming language source code file etc). If the global variable error_one_per_line is set to a non-zero value error_at_line will avoid printing consecutive messages for the same file and line. Repetition which are not directly following each other are not caught. Just like error this function only returns if status is zero. Otherwise exit is called with the non-zero value. If error returns, the global variable error_message_count is incremented by one to keep track of the number of errors reported.

As mentioned above, the error and error_at_line functions can be customized by

defining a variable named error_print_progname.

void (*error_print_progname) (void) If the error_print_progname variable is defined to a non-zero value the function pointed to is called by error or error_at_line. It is expected to print the program name or do something similarly useful. The function is expected to print to the stderr stream and must be able to handle whatever orientation the stream has. The variable is global and shared by all threads.

unsigned int error_message_count The error_message_count variable is incremented whenever one of the functions error or error_at_line returns. The variable is global and shared by all threads.

int error_one_per_line The error_one_per_line variable influences only error_at_line. Normally the error_at_line function creates output for every invocation. If **error_one_per_** line is set to a non-zero value error_at_line keeps track of the last file name and line number for which an error was reported and avoids directly following messages for the same file and line. This variable is global and shared by all threads.

A program which read some input file and reports errors in it could look like this:

```
{
  char *line = NULL;
  size_t len = 0;
  unsigned int lineno = 0;

  error_message_count = 0;
  while (! feof_unlocked (fp))
    {
      ssize_t n = getline (&line, &len, fp);
      if (n <= 0)
        /* End of file or error.  */
        break;
      ++lineno;

      /* Process the line.  */
      ...

      if (Detect error in line)
        error_at_line (0, errval, filename, lineno,
                       "some error text %s", some_variable);
    }

  if (error_message_count != 0)
    error (EXIT_FAILURE, 0, "%u errors found", error_message_count);
}


error and error_at_line are clearly the functions of choice and enable the programmer
```

to write applications which follow the GNU coding standard. The GNU C Library addi- tionally contains functions which are used in BSD for the same purpose. These functions are declared in err.h. It is generally advised to not use these functions. They are included only for compatibility.

void warn (const char *format, . . .) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap i18n | AC-Unsafe corrupt lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The warn function is roughly equivalent to a call like

error (0, errno, format, the parameters) except that the global variables error respects and modifies are not used.

void vwarn (const char *format, va list ap) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap i18n | AC-Unsafe corrupt lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The vwarn function is just like warn except that the parameters for the handling of the format string format are passed in as a value of type va_list.

void warnx (const char *format, . . .) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap | AC-Unsafe corrupt lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2 The warnx function is roughly equivalent to a call like

> error (0, 0, format, the parameters)

except that the global variables error respects and modifies are not used. The dif- ference to warn is that no error number string is printed.

void vwarnx (const char *format, va list ap) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap | AC-Unsafe corrupt lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The vwarnx function is just like warnx except that the parameters for the handling of the format string format are passed in as a value of type va_list.

void err (int status, const char *format, . . .) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap i18n | AC-Unsafe corrupt lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The err function is roughly equivalent to a call like

> error (status, errno, format, the parameters)

except that the global variables error respects and modifies are not used and that the program is exited even if status is zero.

void verr (int status, const char *format, va list ap) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap i18n | AC-Unsafe corrupt lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The verr function is just like err except that the parameters for the handling of the format string format are passed in as a value of type va_list.

void errx (int status, const char *format, . . .) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap | AC-Unsafe corrupt lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The errx function is roughly equivalent to a call like

> error (status, 0, format, the parameters)

except that the global variables error respects and modifies are not used and that the program is exited even if status is zero. The difference to err is that no error number string is printed.

void verrx (int status, const char *format, va list ap) Preliminary: | MT-Safe locale | AS-Unsafe corrupt heap | AC-Unsafe corrupt lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. The verrx function is just like errx except that the parameters for the handling of the format string format are passed in as a value of type va_list.

12 Input/Output on Streams

This chapter describes the functions for creating streams and performing input and output operations on them. As discussed in Chapter 11 [Input/Output Overview], page 245, a stream is a fairly abstract, high-level concept representing a communications channel to a file, device, or process.

3.1 12.1 Streams

For historical reasons, the type of the C data structure that represents a stream is called **FILE** rather than "stream". Since most of the library functions deal with objects of type **FILE ***, sometimes the term file pointer is also used to mean "stream". This leads to unfortunate confusion over terminology in many books on C. This manual, however, is careful to use the terms "file" and "stream" only in the technical sense.

The **FILE** type is declared in the header file **stdio.h**

FILE [Data Type]

This is the data type used to represent stream objects. A FILE object holds all of the internal state information about the connection to the associated file, including such things as the file position indicator and buffering information. Each stream also has error and end-of-file status indicators that can be tested with the ferror and feof functions; see Section 12.15 [End-Of-File and Errors], page 304.

FILE objects are allocated and managed internally by the input/output library functions. Don't try to create your own objects of type FILE; let the library do it. Your programs should deal only with pointers to these objects (that is, FILE * values) rather than the objects themselves.

3.2 12.2 Standard Streams

When the **main** function of your program is invoked, it already has three predefined streams open and available for use. These represent the "standard" input and output channels that have been established for the process.

These streams are declared in the header file **stdio.h**.

FILE * stdin [Variable]

The standard input stream, which is the normal source of input for the program.

FILE * stdout [Variable]

The standard output stream, which is used for normal output from the program.

FILE * stderr [Variable]

The standard error stream, which is used for error messages and diagnostics issued by the program.

On GNU systems, you can specify what files or processes correspond to these streams

using the pipe and redirection facilities provided by the shell. (The primitives shells use to implement these facilities are described in Chapter 14 [File System Interface], page 379.)

Chapter 12: Input/Output on Streams 251

Most other operating systems provide similar mechanisms, but the details of how to use them can vary.

In the GNU C Library, stdin, stdout, and stderr are normal variables which you can set

just like any others. For example, to redirect the standard output to a file, you could do:

fclose (stdout); stdout = fopen ("standard-output-file", "w");

Note however, that in other systems stdin, stdout, and stderr are macros that you cannot

assign to in the normal way. But you can use freopen to get the effect of closing one and reopening it. See Section 12.3 [Opening Streams], page 251.

The three streams stdin, stdout, and stderr are not unoriented at program start (see

Section 12.6 [Streams in Internationalized Applications], page 259).

3.3 12.3 Opening Streams

Opening a file with the fopen function creates a new stream and establishes a connection between the stream and a file. This may involve creating a new file.

Everything described in this section is declared in the header file **stdio.h**.

FILE * fopen (const char **filename*, const char *opentype) [Function]

Preliminary: | MT-Safe | AS-Unsafe heap lock | AC-Unsafe mem fd lock | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The **fopen** function opens a stream for I/O to the file filename, and returns a pointer to the stream.

The opentype argument is a string that controls how the file is opened and specifies attributes of the resulting stream. It must begin with one of the following sequences of characters:

r Open an existing file for reading only.

***w* Open the file for writing only. If the file already exists, it is truncated to** zero length. Otherwise a new file is created.

***a* Open a file for append access; that is, writing at the end of file only. If** the file already exists, its initial contents are unchanged and output to the stream is appended to the end of the file. Otherwise, a new, empty file is created.

***r+* Open an existing file for both reading and writing. The initial contents** of the file are unchanged and the initial file position is at the beginning of the file.

***w+* Open a file for both reading and writing. If the file already exists, it is** truncated to zero length. Otherwise, a new file is created.

***a+* Open or create file for both reading and appending. If the file exists,** its initial contents are unchanged. Otherwise, a new file is created. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

Chapter 12: Input/Output on Streams 252

As you can see, + requests a stream that can do both input and output. When using such a stream, you must call fflush (see Section 12.20 [Stream Buffering], page 312) or a file positioning function such as fseek (see Section 12.18 [File Positioning], page 307) when switching from reading to writing or vice versa. Otherwise, internal buffers might not be emptied properly.

Additional characters may appear after these to specify flags for the call. Always put the mode (*r*, *w+*, etc.) first; that is the only part you are guaranteed will be understood by all systems.

The GNU C Library defines additional characters for use in opentype:

c The file is opened with cancellation in the I/O functions disabled.

***e* The underlying file descriptor will be closed if you use any of the exec...** functions (see Section 26.5 [Executing a File], page 755). (This is equivalent to having set FD_CLOEXEC on that descriptor. See Section 13.13 [File Descriptor Flags], page 363.)

***m* The file is opened and accessed using mmap. This is only supported with** files opened for reading.

***x* Insist on creating a new file—if a file filename already exists, fopen fails** rather than opening it. If you use *x* you are guaranteed that you will not clobber an existing file. This is equivalent to the O_EXCL option to the open function (see Section 13.1 [Opening and Closing Files], page 325).

The *x* modifier is part of ISO C11.

The character *b* in opentype has a standard meaning; it requests a binary stream rather than a text stream. But this makes no difference in POSIX systems (including GNU systems). If both + and *b* are specified, they can appear in either order. See Section 12.17 [Text and Binary Streams], page 306.

If the *opentype* string contains the sequence ,ccs=*STRING* then **STRING** is taken as the name of a coded character set and fopen will mark the stream as wide-oriented with appropriate conversion functions in place to convert from and to the character set **STRING**. Any other stream is opened initially unoriented and the orientation is decided with the first file operation. If the first operation is a wide character operation, the stream is not only marked as wide-oriented, also the conversion functions to convert to the coded character set used for the current locale are loaded. This will not change anymore from this point on even if the locale selected for the **LC_CTYPE** category is changed.

You can have multiple streams (or file descriptors) pointing to the same file open at the

same time. If you do only input, this works straightforwardly, but you must be careful if any

Chapter 12: Input/Output on Streams 253

output streams are included. See Section 13.5 [Dangers of Mixing Streams and Descriptors], page 336. This is equally true whether the streams are in one program (not usual) or in several programs (which can easily happen). It may be advantageous to use the file locking facilities to avoid simultaneous access. See Section 13.15 [File Locks], page 370.

FILE * **fopen64**(const char * **filename**, const char * **opentype**) [Function]

Preliminary: | MT-Safe | AS-Unsafe heap lock | AC-Unsafe mem fd lock | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is similar to fopen but the stream it returns a pointer for is opened using open64. Therefore this stream can be used even on files larger than 231 bytes on 32 bit machines.

Please note that the return type is still FILE *. There is no special FILE type for the LFS interface.

If the sources are compiled with _FILE_OFFSET_BITS == 64 on a 32 bits machine this function is available under the name fopen and so transparently replaces the old interface.

int **FOPEN_MAX** [Macro]

The value of this macro is an integer constant expression that represents the minimum number of streams that the implementation guarantees can be open simultaneously. You might be able to open more than this many streams, but that is not guaranteed. The value of this constant is at least eight, which includes the three standard streams **stdin, stdout,** and **stderr**. In **POSIX.1** systems this value is determined by the **OPEN_MAX** parameter; see Section 32.1 [General Capacity Limits], page 841. In **BSD** and **GNU**, it is controlled by the **RLIMIT_NOFILE** resource limit; see Section 22.2 [Limiting Resource Usage], page 635.

FILE * **freopen** (const char ***filename**, const char ***opentype**,**FILE** ***stream**) [Funcion]

Preliminary: | MT-Safe | AS-Unsafe corrupt | AC-Unsafe corrupt fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is like a combination of **fclose** and **fopen**. It first closes the stream referred to by stream, ignoring any errors that are detected in the process. (Because errors are ignored, you should not use **freopen** on an output stream if you have actually done any output using the stream.) Then the file named by *filename* is opened with mode opentype as for **fopen**, and associated with the same stream object stream.

If the operation fails, a null pointer is returned; otherwise, **freopen** returns stream.

If the operation fails, a null pointer is returned; otherwise, **freopen** returns stream. freopen has traditionally been used to connect a standard stream such as stdin with a file of your own choice. This is useful in programs in which use of a standard stream for certain purposes is hard-coded. In the **GNU C** Library, you can simply close the standard streams and open new ones with fopen. But other systems lack this ability, so using freopen is more portable.

When the sources are compiling with **_FILE_OFFSET_BITS == 64** on a 32 bit machine this function is in fact **freopen64** since the LFS interface replaces transparently the old interface.

Chapter 12: Input/Output on Streams 254

FILE * **freopen64** (const char * **filename**, const char * **opentype**,**FILE** * **stream**) [Function]

Preliminary: | MT-Safe | AS-Unsafe corrupt | AC-Unsafe corrupt fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is similar to freopen. The only difference is that on 32 bit machine the stream returned is able to read beyond the 2**31 bytes limits imposed by the normal interface. It should be noted that the stream pointed to by stream need not be opened using **fopen64** or **freopen64** since its mode is not important for this function.

If the sources are compiled with **_FILE_OFFSET_BITS == 64** on a 32 bits machine this function is available under the name **freopen** and so transparently replaces the old interface.

In some situations it is useful to know whether a given stream is available for reading

or writing. This information is normally not available and would have to be remembered separately. Solaris introduced a few functions to get this information from the stream descriptor and these functions are also available in the **GNU C** Library.

int __freadable (**FILE** * **stream**) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The **__freadable** function determines whether the stream stream was opened to allow reading. In this case the return value is nonzero. For write-only streams the function returns zero.

This function is declared in **stdio_ext.h**.

CHAPTER 4

Inter-Process Communication

This chapter describes the GNU C Library inter-process communication primitives

4.1 Semaphores

The GNU C Library implements the semaphore APIs as defined in POSIX and System V. Semaphores can be used by multiple processes to coordinate shared resources. The following is a complete list of the semaphore functions provided by the GNU C Library.

4.1.1 System V Semaphores

`int semctl(intsemid, intsemnum, intcmd);` Preliminary:|MT-Safe|AS-Safe|AC-Unsafe corrupt/linux|SeeSection 1.2.2.1 POSIX Safety Concepts, page 2.

`int semget(keytkey, intnsems, intsemflg);` Preliminary:|MT-Safe|AS-Safe|AC-Safe|SeeSection 1.2.2.1 POSIX Safety Concepts, page 2.

`int semop(intsemid, struct sembuf *sops, sizetnsops);` Preliminary:|MT-Safe|AS-Safe|AC-Safe|SeeSection 1.2.2.1 POSIX Safety Concepts, page 2.

`int semtimedop(intsemid, struct sembuf *sops, sizetnsops, conststruct timespec *timeout);` Preliminary:|MT-Safe|AS-Safe|AC-Safe|SeeSection 1.2.2.1 POSIX Safety Concepts, page 2.

4.1.2 POSIX Semaphores

`int sem_init(semt *sem, intpshared, unsigned intvalue);` Preliminary:|MT-Safe|AS-Safe|AC-Unsafe corrupt|SeeSection 1.2.2.1 POSIX Safety Concepts, page 2.

`int sem_destroy(semt *sem);` Preliminary:|MT-Safe|AS-Safe|AC-Safe|See Section 1.2.2.1 POSIX Safety Concepts, page 2.

`sem_t *sem_open(const char *name, intoflag, ...);` Preliminary:|MT-Safe|AS-Unsafe init|AC-Unsafe init|See Section 1.2.2.1 POSIX Safety Concepts, page 2.

`int sem_close(semt *sem);` Preliminary:|MT-Safe|AS-Unsafe lock|AC-Unsafe lock|See Section 1.2.2.1 POSIX Safety Concepts, page 2.

`int sem_unlink(const char *name);` Preliminary:|MT-Safe|AS-Unsafe init|AC-Unsafe corrupt|See Section 1.2.2.1 POSIX Safety Concepts, page 2.

int sem_wait(semt *sem); Preliminary:|MT-Safe|AS-Safe|AC-Unsafe corrupt|See Section 1.2.2.1 POSIX Safety Concepts, page 2.

int sem_timedwait(semt *sem, const struct timespec *abstime); Preliminary:|MT-Safe|AS-Safe|AC-Unsafe corrupt|See Section 1.2.2.1 POSIX Safety Concepts, page 2.

int sem_trywait(semt *sem); Preliminary:|MT-Safe|AS-Safe|AC-Safe|See Section 1.2.2.1 POSIX Safety-Concepts, page 2.

int sem_post(semt *sem); Preliminary:|MT-Safe|AS-Safe|AC-Safe|See Section 1.2.2.1 POSIX SafetyConcepts, page 2.

int sem_getvalue(semt *sem, int *sval); Preliminary:|MT-Safe|AS-Safe|AC-Safe|See Section 1.2.2.1 POSIX SafetyConcepts, page 2.

CHAPTER 5

32 System Configuration Parameters

The functions and macros listed in this chapter give information about configuration parameters of the operating system—for example, capacity limits, presence of optional POSIX features, and the default path for executable files (see Section 32.12 [String-Valued Parameters], page 859).

5.1 32.1 General Capacity Limits

The POSIX.1 and POSIX.2 standards specify a number of parameters that describe capacity limitations of the system. These limits can be fixed constants for a given operating system, or they can vary from machine to machine. For example, some limit values may be configurable by the system administrator, either at run time or by rebuilding the kernel, and this should not require recompiling application programs. Each of the following limit parameters has a macro that is defined in limits.h only if the system has a fixed, uniform limit for the parameter in question. If the system allows different file systems or files to have different limits, then the macro is undefined; use sysconf to find out the limit that applies at a particular time on a particular machine. See Section 32.4 [Using sysconf], page 844. Each of these parameters also has another macro, with a name starting with '_POSIX', which gives the lowest value that the limit is allowed to have on any POSIX system. See Section 32.5 [Minimum Values for General Capacity Limits], page 852.

int ARG_MAX [Macro] If defined, the unvarying maximum combined length of the argv and environ arguments

that can be passed to the exec functions.

int CHILD_MAX [Macro] If defined, the unvarying maximum number of processes that can exist with the same

real user ID at any one time. In BSD and GNU, this is controlled by the **RLIMIT_** NPROC resource limit; see Section 22.2 [Limiting Resource Usage], page 635.

int OPEN_MAX [Macro] If defined, the unvarying maximum number of files that a single process can have open

simultaneously. In BSD and GNU, this is controlled by the RLIMIT_NOFILE resource limit; see Section 22.2 [Limiting Resource Usage], page 635.

int STREAM_MAX [Macro] If defined, the unvarying maximum number of streams that a single process can have

open simultaneously. See Section 12.3 [Opening Streams], page 251.

int TZNAME_MAX [Macro] If defined, the unvarying maximum length of a time zone name. See Section 21.4.8

[Functions and Variables for Time Zones], page 627. These limit macros are always defined in limits.h

int NGROUPS_MAX [Macro] The maximum number of supplementary group IDs that one process can have.

The value of this macro is actually a lower bound for the maximum. That is, you can count on being able to have that many supplementary group IDs, but a particular machine might let you have even more. You can use sysconf to see whether a particular machine will let you have more (see Section 32.4 [Using sysconf], page 844).

ssize_t SSIZE_MAX [Macro] The largest value that can fit in an object of type ssize_t. Effectively, this is the

limit on the number of bytes that can be read or written in a single operation. This macro is defined in all POSIX systems because this limit is never configurable.

int RE_DUP_MAX [Macro] The largest number of repetitions you are guaranteed is allowed in the construct

'{min,max}' in a regular expression. The value of this macro is actually a lower bound for the maximum. That is, you can count on being able to have that many repetitions, but a particular machine might let you have even more. You can use sysconf to see whether a particular machine will let you have more (see Section 32.4 [Using sysconf], page 844). And even the value that sysconf tells you is just a lower bound—larger values might work. This macro is defined in all POSIX.2 systems, because POSIX.2 says it should always be defined even if there is no specific imposed limit.

5.2 32.2 Overall System Options

POSIX defines certain system-specific options that not all POSIX systems support. Since these options are provided in the kernel, not in the library, simply using the GNU C Library does not guarantee any of these features is supported; it depends on the system you are using. You can test for the availability of a given option using the macros in this section, together with the function sysconf. The macros are defined only if you include unistd.h. For the following macros, if the macro is defined in unistd.h, then the option is supported. Otherwise, the option may or may not be supported; use sysconf to find out. See Section 32.4 [Using sysconf], page 844.

int _POSIX_JOB_CONTROL [Macro] If this symbol is defined, it indicates that the system supports job control. Otherwise,

the implementation behaves as if all processes within a session belong to a single process group. See Chapter 28 [Job Control], page 765.

int _POSIX_SAVED_IDS [Macro] If this symbol is defined, it indicates that the system remembers the effective user and

group IDs of a process before it executes an executable file with the set-user-ID or setgroup-ID bits set, and that explicitly changing the effective user or group IDs back to these values is permitted. If this option is not defined, then if a nonprivileged process changes its effective user or group ID to the real user or group ID of the process, it can't change it back again. See Section 30.8 [Enabling and Disabling Setuid Access], page 800.

For the following macros, if the macro is defined in unistd.h, then its value indicates whether the option is supported. A value of -1 means no, and any other value means yes. If the macro is not defined, then the option may or may not be supported; use sysconf to find out. See Section 32.4 [Using sysconf], page 844.

int _POSIX2_C_DEV [Macro] If this symbol is defined, it indicates that the system has the POSIX.2 C compiler

command, c89. The GNU C Library always defines this as 1, on the assumption that you would not have installed it if you didn't have a C compiler.

int _POSIX2_FORT_DEV [Macro] If this symbol is defined, it indicates that the system has the POSIX.2 Fortran compiler

command, fort77. The GNU C Library never defines this, because we don't know what the system has.

int _POSIX2_FORT_RUN [Macro] If this symbol is defined, it indicates that the system has the POSIX.2 asa command

to interpret Fortran carriage control. The GNU C Library never defines this, because we don't know what the system has.

int _POSIX2_LOCALEDEF [Macro] If this symbol is defined, it indicates that the system has the POSIX.2 localedef

command. The GNU C Library never defines this, because we don't know what the system has.

int _POSIX2_SW_DEV [Macro] If this symbol is defined, it indicates that the system has the POSIX.2 commands ar,

make, and strip. The GNU C Library always defines this as 1, on the assumption that you had to have ar and make to install the library, and it's unlikely that strip would be absent when those are present.

5.3 32.3 Which Version of POSIX is Supported

long int _POSIX_VERSION [Macro] This constant represents the version of the POSIX.1 standard to which the implementation

conforms. For an implementation conforming to the 1995 POSIX.1 standard, the value is the integer 199506L.

_POSIX_VERSION is always defined (in unistd.h) in any POSIX system. Usage Note: Don't try to test whether the system supports POSIX by including unistd.h and then checking whether _POSIX_VERSION is defined. On a non-POSIX system, this will probably fail because there is no unistd.h. We do not know of any way you can reliably test at compilation time whether your target system supports POSIX or whether unistd.h exists.

long int _POSIX2_C_VERSION [Macro] This constant represents the version of the POSIX.2 standard which the library and

system kernel support. We don't know what value this will be for the first version of the POSIX.2 standard, because the value is based on the year and month in which the standard is officially adopted.

The value of this symbol says nothing about the utilities installed on the system. Usage Note: You can use this macro to tell whether a POSIX.1 system library supports POSIX.2 as well. Any POSIX.1 system contains unistd.h, so include that file and then test defined (_POSIX2_C_VERSION).

5.4 32.4 Using sysconf

When your system has configurable system limits, you can use the sysconf function to find out the value that applies to any particular machine. The function and the associated parameter constants are declared in the header file unistd.h.

5.5 32.4.1 Definition of sysconf

long int sysconf (int parameter) [Function]

Preliminary: | MT-Safe env | AS-Unsafe lock heap | AC-Unsafe lock mem fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. This function is used to inquire about runtime system parameters. The parameter argument should be one of the '_SC_' symbols listed below. The normal return value from sysconf is the value you requested. A value of -1 is returned both if the implementation does not impose a limit, and in case of an error. The following errno error conditions are defined for this function: EINVAL The value of the parameter is invalid.

5.6 32.4.2 Constants for sysconf Parameters

Here are the symbolic constants for use as the parameter argument to sysconf. The values are all integer constants (more specifically, enumeration type values).

_SC_ARG_MAX Inquire about the parameter corresponding to ARG_MAX.

_SC_CHILD_MAX Inquire about the parameter corresponding to CHILD_MAX.

_SC_OPEN_MAX Inquire about the parameter corresponding to OPEN_MAX.

_SC_STREAM_MAX Inquire about the parameter corresponding to STREAM_MAX.

_SC_TZNAME_MAX Inquire about the parameter corresponding to TZNAME_MAX.

_SC_NGROUPS_MAX Inquire about the parameter corresponding to NGROUPS_MAX.

_SC_JOB_CONTROL Inquire about the parameter corresponding to _POSIX_JOB_CONTROL.

_SC_SAVED_IDS Inquire about the parameter corresponding to _POSIX_SAVED_IDS.

_SC_VERSION Inquire about the parameter corresponding to _POSIX_VERSION.

_SC_CLK_TCK Inquire about the number of clock ticks per second; see Section 21.3.1 [CPU Time Inquiry], page 600. The corresponding parameter CLK_TCK is obsolete.

_SC_CHARCLASS_NAME_MAX Inquire about the parameter corresponding to maximal length allowed for a character class name in an extended locale specification. These extensions are not yet standardized and so this option is not standardized as well.

_SC_REALTIME_SIGNALS Inquire about the parameter corresponding to _POSIX_REALTIME_SIGNALS.

_SC_PRIORITY_SCHEDULING Inquire about the parameter corresponding to _POSIX_PRIORITY_SCHEDULING.

_SC_TIMERS Inquire about the parameter corresponding to _POSIX_TIMERS.

_SC_ASYNCHRONOUS_IO Inquire about the parameter corresponding to _POSIX_ASYNCHRONOUS_IO.

_SC_PRIORITIZED_IO Inquire about the parameter corresponding to _POSIX_PRIORITIZED_IO.

_SC_SYNCHRONIZED_IO Inquire about the parameter corresponding to _POSIX_SYNCHRONIZED_IO.

_SC_FSYNC Inquire about the parameter corresponding to _POSIX_FSYNC.

_SC_MAPPED_FILES Inquire about the parameter corresponding to _POSIX_MAPPED_FILES.

_SC_MEMLOCK Inquire about the parameter corresponding to _POSIX_MEMLOCK.

_SC_MEMLOCK_RANGE Inquire about the parameter corresponding to _POSIX_MEMLOCK_RANGE.

_SC_MEMORY_PROTECTION Inquire about the parameter corresponding to _POSIX_MEMORY_PROTECTION.

_SC_MESSAGE_PASSING Inquire about the parameter corresponding to _POSIX_MESSAGE_PASSING.

_SC_SEMAPHORES Inquire about the parameter corresponding to _POSIX_SEMAPHORES.

_SC_SHARED_MEMORY_OBJECTS Inquire about the parameter corresponding to

_POSIX_SHARED_MEMORY_OBJECTS.

_SC_AIO_LISTIO_MAX Inquire about the parameter corresponding to _POSIX_AIO_LISTIO_MAX.

_SC_AIO_MAX Inquire about the parameter corresponding to _POSIX_AIO_MAX.

_SC_AIO_PRIO_DELTA_MAX Inquire the value by which a process can decrease its asynchronous I/O priority level from its own scheduling priority. This corresponds to the run-time invariant value AIO_PRIO_DELTA_MAX.

_SC_DELAYTIMER_MAX Inquire about the parameter corresponding to _POSIX_DELAYTIMER_MAX.

_SC_MQ_OPEN_MAX Inquire about the parameter corresponding to _POSIX_MQ_OPEN_MAX.

_SC_MQ_PRIO_MAX Inquire about the parameter corresponding to _POSIX_MQ_PRIO_MAX.

_SC_RTSIG_MAX Inquire about the parameter corresponding to _POSIX_RTSIG_MAX.

_SC_SEM_NSEMS_MAX Inquire about the parameter corresponding to _POSIX_SEM_NSEMS_MAX.

_SC_SEM_VALUE_MAX Inquire about the parameter corresponding to _POSIX_SEM_VALUE_MAX.

_SC_SIGQUEUE_MAX Inquire about the parameter corresponding to _POSIX_SIGQUEUE_MAX.

_SC_TIMER_MAX Inquire about the parameter corresponding to _POSIX_TIMER_MAX.

_SC_PII Inquire about the parameter corresponding to _POSIX_PII.

_SC_PII_XTI Inquire about the parameter corresponding to _POSIX_PII_XTI.

_SC_PII_SOCKET Inquire about the parameter corresponding to _POSIX_PII_SOCKET.

_SC_PII_INTERNET Inquire about the parameter corresponding to _POSIX_PII_INTERNET.

_SC_PII_OSI Inquire about the parameter corresponding to _POSIX_PII_OSI.

_SC_SELECT Inquire about the parameter corresponding to _POSIX_SELECT.

_SC_UIO_MAXIOV Inquire about the parameter corresponding to _POSIX_UIO_MAXIOV.

_SC_PII_INTERNET_STREAM Inquire about the parameter corresponding to _POSIX_PII_INTERNET_STREAM.

_SC_PII_INTERNET_DGRAM Inquire about the parameter corresponding to _POSIX_PII_INTERNET_DGRAM.

_SC_PII_OSI_COTS Inquire about the parameter corresponding to _POSIX_PII_OSI_COTS.

_SC_PII_OSI_CLTS Inquire about the parameter corresponding to _POSIX_PII_OSI_CLTS.

_SC_PII_OSI_M Inquire about the parameter corresponding to _POSIX_PII_OSI_M.

_SC_T_IOV_MAX Inquire the value of the value associated with the T_IOV_MAX variable.

_SC_THREADS Inquire about the parameter corresponding to _POSIX_THREADS.

_SC_THREAD_SAFE_FUNCTIONS Inquire about the parameter corresponding to

_POSIX_THREAD_SAFE_FUNCTIONS.

_SC_GETGR_R_SIZE_MAX Inquire about the parameter corresponding to _POSIX_GETGR_R_SIZE_MAX.

_SC_GETPW_R_SIZE_MAX Inquire about the parameter corresponding to _POSIX_GETPW_R_SIZE_MAX.

_SC_LOGIN_NAME_MAX Inquire about the parameter corresponding to _POSIX_LOGIN_NAME_MAX.

_SC_TTY_NAME_MAX Inquire about the parameter corresponding to _POSIX_TTY_NAME_MAX.

_SC_THREAD_DESTRUCTOR_ITERATIONS Inquire about the parameter corresponding to _POSIX_THREAD_DESTRUCTOR_ ITERATIONS.

_SC_THREAD_KEYS_MAX Inquire about the parameter corresponding to _POSIX_THREAD_KEYS_MAX.

_SC_THREAD_STACK_MIN Inquire about the parameter corresponding to _POSIX_THREAD_STACK_MIN.

_SC_THREAD_THREADS_MAX Inquire about the parameter corresponding to _POSIX_THREAD_THREADS_MAX.

_SC_THREAD_ATTR_STACKADDR Inquire about the parameter corresponding to a _POSIX_THREAD_ATTR_STACKADDR.

_SC_THREAD_ATTR_STACKSIZE Inquire about the parameter corresponding to _POSIX_THREAD_ATTR_STACKSIZE.

_SC_THREAD_PRIORITY_SCHEDULING Inquire about the parameter corresponding to _POSIX_THREAD_PRIORITY_ SCHEDULING.

_SC_THREAD_PRIO_INHERIT Inquire about the parameter corresponding to _POSIX_THREAD_PRIO_INHERIT.

_SC_THREAD_PRIO_PROTECT Inquire about the parameter corresponding to _POSIX_THREAD_PRIO_PROTECT.

_SC_THREAD_PROCESS_SHARED Inquire about the parameter corresponding to _POSIX_THREAD_PROCESS_SHARED.

_SC_2_C_DEV Inquire about whether the system has the POSIX.2 C compiler command, c89.

_SC_2_FORT_DEV Inquire about whether the system has the POSIX.2 Fortran compiler command, fort77.

_SC_2_FORT_RUN Inquire about whether the system has the POSIX.2 asa command to interpret Fortran carriage control.

_SC_2_LOCALEDEF Inquire about whether the system has the POSIX.2 localedef command.

_SC_2_SW_DEV Inquire about whether the system has the POSIX.2 commands ar, make, and strip.

_SC_BC_BASE_MAX Inquire about the maximum value of obase in the bc utility.

_SC_BC_DIM_MAX Inquire about the maximum size of an array in the bc utility.

_SC_BC_SCALE_MAX Inquire about the maximum value of scale in the bc utility.

_SC_BC_STRING_MAX Inquire about the maximum size of a string constant in the bc utility.

_SC_COLL_WEIGHTS_MAX Inquire about the maximum number of weights that can necessarily be used in defining the collating sequence for a locale.

_SC_EXPR_NEST_MAX Inquire about the maximum number of expressions nested within parentheses when using the expr utility.

_SC_LINE_MAX Inquire about the maximum size of a text line that the POSIX.2 text utilities can handle.

_SC_EQUIV_CLASS_MAX Inquire about the maximum number of weights that can be assigned to an entry of the LC_COLLATE category 'order' keyword in a locale definition. The GNU C Library does not presently support locale definitions.

SC_EQUIV_CLASS_MAX Inquire about the maximum number of weights that can be assigned to an entry of the LC_COLLATE category 'order' keyword in a locale definition. The GNU C Library does not presently support locale definitions.

_SC_VERSION Inquire about the version number of POSIX.1 that the library and kernel support.

_SC_2_VERSION Inquire about the version number of POSIX.2 that the system utilities support.

_SC_PAGESIZE Inquire about the virtual memory page size of the machine. getpagesize returns the same value (see Section 22.4.2 [How to get information about the memory subsystem?], page 651).

_SC_NPROCESSORS_CONF Inquire about the number of configured processors.

_SC_NPROCESSORS_ONLN Inquire about the number of processors online.

_SC_PHYS_PAGES Inquire about the number of physical pages in the system.

_SC_AVPHYS_PAGES Inquire about the number of available physical pages in the system.

_SC_ATEXIT_MAX Inquire about the number of functions which can be registered as termination functions for atexit; see Section 25.7.3 [Cleanups on Exit], page 749.

_SC_XOPEN_VERSION Inquire about the parameter corresponding to _XOPEN_VERSION.

_SC_XOPEN_XCU_VERSION Inquire about the parameter corresponding to _XOPEN_XCU_VERSION.

_SC_XOPEN_UNIX Inquire about the parameter corresponding to _XOPEN_UNIX.

_SC_XOPEN_REALTIME Inquire about the parameter corresponding to _XOPEN_REALTIME.

_SC_XOPEN_REALTIME_THREADS Inquire about the parameter corresponding to _XOPEN_REALTIME_THREADS.

_SC_XOPEN_LEGACY Inquire about the parameter corresponding to _XOPEN_LEGACY.

_SC_XOPEN_CRYPT Inquire about the parameter corresponding to _XOPEN_CRYPT.

_SC_XOPEN_ENH_I18N Inquire about the parameter corresponding to _XOPEN_ENH_I18N.

_SC_XOPEN_SHM Inquire about the parameter corresponding to _XOPEN_SHM.

_SC_XOPEN_XPG2 Inquire about the parameter corresponding to _XOPEN_XPG2.

_SC_XOPEN_XPG3 Inquire about the parameter corresponding to _XOPEN_XPG3.

_SC_XOPEN_XPG4 Inquire about the parameter corresponding to _XOPEN_XPG4.

_SC_CHAR_BIT Inquire about the number of bits in a variable of type char.

_SC_CHAR_MAX Inquire about the maximum value which can be stored in a variable of type char.

_SC_CHAR_MIN Inquire about the minimum value which can be stored in a variable of type char.

_SC_INT_MAX Inquire about the maximum value which can be stored in a variable of type int.

_SC_INT_MIN Inquire about the minimum value which can be stored in a variable of type int.

_SC_LONG_BIT Inquire about the number of bits in a variable of type long int.

_SC_WORD_BIT Inquire about the number of bits in a variable of a register word.

_SC_MB_LEN_MAX Inquire the maximum length of a multi-byte representation of a wide character value.

_SC_NZERO Inquire about the value used to internally represent the zero priority level for the process execution.

SC_SSIZE_MAX Inquire about the maximum value which can be stored in a variable of type ssize_t.

_SC_SCHAR_MAX Inquire about the maximum value which can be stored in a variable of type signed char.

_SC_SCHAR_MIN Inquire about the minimum value which can be stored in a variable of type signed char.

_SC_SHRT_MAX Inquire about the maximum value which can be stored in a variable of type short int.

_SC_SHRT_MIN Inquire about the minimum value which can be stored in a variable of type short int.

_SC_UCHAR_MAX Inquire about the maximum value which can be stored in a variable of type unsigned char.

_SC_UINT_MAX Inquire about the maximum value which can be stored in a variable of type unsigned int.

_SC_ULONG_MAX Inquire about the maximum value which can be stored in a variable of type unsigned long int.

_SC_USHRT_MAX Inquire about the maximum value which can be stored in a variable of type unsigned short int.

_SC_NL_ARGMAX Inquire about the parameter corresponding to NL_ARGMAX.

_SC_NL_LANGMAX Inquire about the parameter corresponding to NL_LANGMAX.

_SC_NL_MSGMAX Inquire about the parameter corresponding to NL_MSGMAX.

_SC_NL_NMAX Inquire about the parameter corresponding to NL_NMAX.

_SC_NL_SETMAX Inquire about the parameter corresponding to NL_SETMAX.

_SC_NL_TEXTMAX Inquire about the parameter corresponding to NL_TEXTMAX.

5.7 32.4.3 Examples of sysconf

We recommend that you first test for a macro definition for the parameter you are interested in, and call sysconf only if the macro is not defined. For example, here is how to test whether job control is supported:

Here is how to get the value of a numeric limit:

```
int
get_child_max ()
{
#ifdef CHILD_MAX
    return CHILD_MAX;
#else
    int value = sysconf (_SC_CHILD_MAX);
    if (value < 0)
        fatal (strerror (errno));
        return value;
#endif
}
```

5.8 32.5 Minimum Values for General Capacity Limits

Here are the names for the POSIX minimum upper bounds for the system limit parameters. The significance of these values is that you can safely push to these limits without checking whether the particular system you are using can go that far.

_POSIX_AIO_LISTIO_MAX The most restrictive limit permitted by POSIX for the maximum number of I/O operations that can be specified in a list I/O call. The value of this constant is 2; thus you can add up to two new entries of the list of outstanding operations.

_POSIX_AIO_MAX The most restrictive limit permitted by POSIX for the maximum number of outstanding asynchronous I/O operations. The value of this constant is 1. So you cannot expect that you can issue more than one operation and immediately continue with the normal work, receiving the notifications asynchronously.

_POSIX_ARG_MAX The value of this macro is the most restrictive limit permitted by POSIX for the maximum combined length of the argv and environ arguments that can be passed to the exec functions. Its value is 4096.

_POSIX_CHILD_MAX The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of simultaneous processes per real user ID. Its value is 6.

_POSIX_NGROUPS_MAX The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of supplementary group IDs per process. Its value is 0.

_POSIX_OPEN_MAX The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of files that a single process can have open simultaneously. Its value is 16.

_POSIX_SSIZE_MAX The value of this macro is the most restrictive limit permitted by POSIX for the maximum value that can be stored in an object of type ssize_t. Its value is 32767.

_POSIX_STREAM_MAX The value of this macro is the most restrictive limit permitted by POSIX for the maximum number of streams that a single process can have open simultaneously. Its value is 8.

_POSIX_TZNAME_MAX The value of this macro is the most restrictive limit permitted by POSIX for the maximum length of a time zone name. Its value is 3.

_POSIX2_RE_DUP_MAX The value of this macro is the most restrictive limit permitted by POSIX for the numbers used in the '{min,max}' construct in a regular expression. Its value is 255.

5.9 32.6 Limits on File System Capacity

The POSIX.1 standard specifies a number of parameters that describe the limitations of the file system. It's possible for the system to have a fixed, uniform limit for a parameter, but this isn't the usual case. On most systems, it's possible for different file systems (and, for some parameters, even different files) to have different maximum limits.

For example, this is very likely if you use NFS to mount some of the file systems from other machines. Each of the following macros is defined in limits.h only if the system has a fixed, uniform limit for the parameter in question. If the system allows different file systems or files to have different limits, then the macro is undefined; use pathconf or fpathconf to find out the limit that applies to a particular file. See Section 32.9 [Using pathconf], page 856. Each parameter also has another macro, with a name starting with '_POSIX', which gives the lowest value that the limit is allowed to have on any POSIX system. See Section 32.8 [Minimum Values for File System Limits], page 855.

int LINK_MAX [Macro] The uniform system limit (if any) for the number of names for a given file. See Section 14.4 [Hard Links], page 394.

int MAX_CANON [Macro] The uniform system limit (if any) for the amount of text in a line of input when input editing is enabled. See Section 17.3 [Two Styles of Input: Canonical or Not], page 480.

int MAX_INPUT [Macro] The uniform system limit (if any) for the total number of characters typed ahead as input. See Section 17.2 [I/O Queues], page 480.

int NAME_MAX [Macro] The uniform system limit (if any) for the length of a file name component, not including the terminating null character. Portability Note: On some systems, the GNU C Library defines NAME_MAX, but does not actually enforce this limit.

int PATH_MAX [Macro] The uniform system limit (if any) for the length of an entire file name (that is, the argument given to system calls such as open), including the terminating null character. Portability Note: The GNU C Library does not enforce this limit even if PATH_MAX is defined.

int PIPE_BUF [Macro] The uniform system limit (if any) for the number of bytes that can be written atomically to a pipe. If multiple processes are writing to the same pipe simultaneously, output from different processes might be interleaved in chunks of this size. See Chapter 15 [Pipes and FIFOs], page 426. These are alternative macro names for some of the same information.

int MAXNAMLEN [Macro] This is the BSD name for NAME_MAX. It is defined in dirent.h.

int FILENAME_MAX [Macro] The value of this macro is an integer constant expression that represents the maximum length of a file name string. It is defined in stdio.h. Unlike PATH_MAX, this macro is defined even if there is no actual limit imposed. In such a case, its value is typically a very large number. This is always the case on GNU/Hurd systems. Usage Note: Don't use FILENAME_MAX as the size of an array in which to store a file name! You can't possibly make an array that big! Use dynamic allocation (see Section 3.2 [Allocating Storage For Program Data], page 40) instead.

5.10 32.7 Optional Features in File Support

POSIX defines certain system-specific options in the system calls for operating on files. Some systems support these options and others do not. Since these options are provided in the kernel, not in the library, simply using the GNU C Library does not guarantee that any of these features is supported; it depends on the system you are using. They can also vary between file systems on a single machine. This section describes the macros you can test to determine whether a particular option is supported on your machine. If a given macro is defined in unistd.h, then its value says whether the corresponding feature is supported. (A value of -1 indicates no; any other value indicates yes.) If the macro is undefined, it means particular files may or may not support the feature. Since all the machines that support the GNU C Library also support NFS, one can never make a general statement about whether all file systems support the _POSIX_CHOWN_ RESTRICTED and _POSIX_NO_TRUNC features. So these names are never defined as macros in the GNU C Library.

int _POSIX_CHOWN_RESTRICTED [Macro] If this option is in effect, the chown function is restricted so that the only changes permitted to nonprivileged processes is to change the group owner of a file to either be the effective group ID of the process, or one of its supplementary group IDs. See Section 14.9.4 [File Owner], page 408.

int _POSIX_NO_TRUNC [Macro] If this option is in effect, file name components longer than NAME_MAX generate an ENAMETOOLONG error. Otherwise, file name components that are too long are silently truncated.

unsigned char _POSIX_VDISABLE [Macro] This option is only meaningful for files that are terminal devices. If it is enabled, then handling for special control characters can be disabled individually. See Section 17.4.9 [Special Characters], page 492. If one of these macros is undefined, that means that the option might be in effect for some files and not for others. To inquire about a particular file, call pathconf or fpathconf. See Section 32.9 [Using pathconf], page 856.

5.11 32.8 Minimum Values for File System Limits

Here are the names for the POSIX minimum upper bounds for some of the above parameters. The significance of these values is that you can safely push to these limits without checking whether the particular system you are using can go that far. In most cases GNU systems do not have these strict limitations. The actual limit should be requested if necessary.

_POSIX_LINK_MAX The most restrictive limit permitted by POSIX for the maximum value of a file's link count. The value of this constant is 8; thus, you can always make up to eight names for a file without running into a system limit.

_POSIX_MAX_CANON The most restrictive limit permitted by POSIX for the maximum number of bytes in a canonical input line from a terminal device. The value of this constant is 255.

_POSIX_MAX_INPUT The most restrictive limit permitted by POSIX for the maximum number of bytes in a terminal device input queue (or typeahead buffer). See Section 17.4.4 [Input Modes], page 484. The value of this constant is 255.

_POSIX_NAME_MAX The most restrictive limit permitted by POSIX for the maximum number of bytes in a file name component. The value of this constant is 14.

_POSIX_PATH_MAX The most restrictive limit permitted by POSIX for the maximum number of bytes in a file name. The value of this constant is 256.

_POSIX_PIPE_BUF The most restrictive limit permitted by POSIX for the maximum number of bytes that can be written atomically to a pipe. The value of this constant is 512.

SYMLINK_MAX Maximum number of bytes in a symbolic link.

POSIX_REC_INCR_XFER_SIZE Recommended increment for file transfer sizes between the POSIX_REC_MIN_XFER_SIZE and POSIX_REC_MAX_XFER_SIZE values.

POSIX_REC_MAX_XFER_SIZE Maximum recommended file transfer size.

POSIX_REC_MIN_XFER_SIZE Minimum recommended file transfer size.

POSIX_REC_XFER_ALIGN Recommended file transfer buffer alignment.

5.12 32.9 Using pathconf

When your machine allows different files to have different values for a file system parameter, you can use the functions in this section to find out the value that applies to any particular file. These functions and the associated constants for the parameter argument are declared in the header file unistd.h. `long int pathconf (const char filename, int parameter) [Function]` Preliminary: | MT-Safe | AS-Unsafe lock heap | AC-Unsafe lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2. This function is used to inquire about the limits that apply to the file named filename. The parameter argument should be one of the '_PC_' constants listed below. The normal return value from pathconf is the value you requested. A value of -1 is returned both if the implementation does not impose a limit, and in case of an error. In the former case, errno is not set, while in the latter case, errno is set to indicate the cause of the problem. So the only way to use this function robustly is to store 0 into errno just before calling it. Besides the usual file name errors (see Section 11.2.3 [File Name Errors], page 248), the following error condition is defined for this function:

EINVAL The value of parameter is invalid, or the implementation doesn't support the parameter for the specific file.

long int fpathconf (int filedes, int parameter) [Function] Preliminary: | MT-Safe | AS-Unsafe lock heap | AC-Unsafe lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This is just like pathconf except that an open file descriptor is used to specify the file for which information is requested, instead of a file name. The following errno error conditions are defined for this function: EBADF The filedes argument is not a valid file descriptor. EINVAL The value of parameter is invalid, or the implementation doesn't support the parameter for the specific file

Here are the symbolic constants that you can use as the parameter argument to pathconf and fpathconf. The values are all integer constants.

_PC_LINK_MAX Inquire about the value of LINK_MAX.

_PC_MAX_CANON Inquire about the value of MAX_CANON.

_PC_MAX_INPUT Inquire about the value of MAX_INPUT.

_PC_NAME_MAX Inquire about the value of NAME_MAX.

_PC_PATH_MAX Inquire about the value of PATH_MAX.

_PC_PIPE_BUF Inquire about the value of PIPE_BUF.

_PC_CHOWN_RESTRICTED Inquire about the value of _POSIX_CHOWN_RESTRICTED.

_PC_NO_TRUNC Inquire about the value of _POSIX_NO_TRUNC.

_PC_VDISABLE Inquire about the value of _POSIX_VDISABLE.

_PC_SYNC_IO Inquire about the value of _POSIX_SYNC_IO.

_PC_ASYNC_IO Inquire about the value of _POSIX_ASYNC_IO.

_PC_PRIO_IO Inquire about the value of _POSIX_PRIO_IO.

_PC_FILESIZEBITS Inquire about the availability of large files on the filesystem.

_PC_REC_INCR_XFER_SIZE Inquire about the value of POSIX_REC_INCR_XFER_SIZE.

_PC_REC_MAX_XFER_SIZE Inquire about the value of POSIX_REC_MAX_XFER_SIZE.

_PC_REC_MIN_XFER_SIZE Inquire about the value of POSIX_REC_MIN_XFER_SIZE.

_PC_REC_XFER_ALIGN Inquire about the value of POSIX_REC_XFER_ALIGN.

Portability Note: On some systems, the GNU C Library does not enforce _PC_NAME_MAX or _PC_PATH_MAX limits.

5.13 32.10 Utility Program Capacity Limits

The POSIX.2 standard specifies certain system limits that you can access through sysconf that apply to utility behavior rather than the behavior of the library or the operating system. The GNU C Library defines macros for these limits, and sysconf returns values for them if you ask; but these values convey no meaningful information. They are simply the smallest values that POSIX.2 permits.

int BC_BASE_MAX [Macro] The largest value of obase that the bc utility is guaranteed to support.

int BC_DIM_MAX [Macro] The largest number of elements in one array that the bc utility is guaranteed to support.

int BC_SCALE_MAX [Macro] The largest value of scale that the bc utility is guaranteed to support.

int BC_STRING_MAX [Macro] The largest number of characters in one string constant that the bc utility is guaranteed to support.

int COLL_WEIGHTS_MAX [Macro] The largest number of weights that can necessarily be used in defining the collating sequence for a locale.

int EXPR_NEST_MAX [Macro] The maximum number of expressions that can be nested within parenthesis by the expr utility.

int LINE_MAX [Macro] The largest text line that the text-oriented POSIX.2 utilities can support. (If you are using the GNU versions of these utilities, then there is no actual limit except that imposed by the available virtual memory, but there is no way that the library can tell you this.)

int EQUIV_CLASS_MAX [Macro] The maximum number of weights that can be assigned to an entry of the LC_COLLATE category 'order' keyword in a locale definition. The GNU C Library does not presently support locale definitions.

5.14 32.11 Minimum Values for Utility Limits

_POSIX2_BC_BASE_MAX The most restrictive limit permitted by POSIX.2 for the maximum value of obase in the bc utility. Its value is 99.

_POSIX2_BC_DIM_MAX The most restrictive limit permitted by POSIX.2 for the maximum size of an array in the bc utility. Its value is 2048.

_POSIX2_BC_SCALE_MAX The most restrictive limit permitted by POSIX.2 for the maximum value of scale in the bc utility. Its value is 99.

_POSIX2_BC_STRING_MAX The most restrictive limit permitted by POSIX.2 for the maximum size of a string constant in the bc utility. Its value is 1000.

_POSIX2_COLL_WEIGHTS_MAX The most restrictive limit permitted by POSIX.2 for the maximum number of weights that can necessarily be used in defining the collating sequence for a locale. Its value is 2.

_POSIX2_EXPR_NEST_MAX The most restrictive limit permitted by POSIX.2 for the maximum number of expressions nested within parenthesis when using the expr utility. Its value is 32.

_POSIX2_LINE_MAX The most restrictive limit permitted by POSIX.2 for the maximum size of a text line that the text utilities can handle. Its value is 2048.

_POSIX2_EQUIV_CLASS_MAX The most restrictive limit permitted by POSIX.2 for the maximum number of weights that can be assigned to an entry of the LC_COLLATE category 'order' keyword in a locale definition. Its value is 2. The GNU C Library does not presently support locale definitions.

5.15 32.12 String-Valued Parameters

POSIX.2 defines a way to get string-valued parameters from the operating system with the function confstr:

size_t confstr (int parameter, char buf, size t len) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function reads the value of a string-valued system parameter, storing the string into len bytes of memory space starting at buf. The parameter argument should be one of the '_CS_' symbols listed below. The normal return value from confstr is the length of the string value that you asked for. If you supply a null pointer for buf, then confstr does not try to store the string; it just returns its length. A value of 0 indicates an error. If the string you asked for is too long for the buffer (that is, longer than len - 1), then confstr stores just that much (leaving room for the terminating null character). You can tell that this has happened because confstr returns a value greater than or equal to len. The following errno error conditions are defined for this function: EINVAL The value of the parameter is invalid.

Currently there is just one parameter you can read with confstr:

_CS_PATH This parameter's value is the recommended default path for searching for executable files. This is the path that a user has by default just after logging in.

_CS_LFS_CFLAGS The returned string specifies which additional flags must be given to the C compiler if a source is compiled using the _LARGEFILE_SOURCE feature select macro; see Section 1.3.4 [Feature Test Macros], page 15.

_CS_LFS_LDFLAGS The returned string specifies which additional flags must be given to the linker if a source is compiled using the _LARGEFILE_SOURCE feature select macro; see

Section 1.3.4 [Feature Test Macros], page 15.

_CS_LFS_LIBS The returned string specifies which additional libraries must be linked to the application if a source is compiled using the _LARGEFILE_SOURCE feature select macro; see Section 1.3.4 [Feature Test Macros], page 15.

_CS_LFS_LINTFLAGS The returned string specifies which additional flags must be given to the lint tool if a source is compiled using the _LARGEFILE_SOURCE feature select macro; see Section 1.3.4 [Feature Test Macros], page 15.

_CS_LFS64_CFLAGS The returned string specifies which additional flags must be given to the C compiler if a source is compiled using the _LARGEFILE64_SOURCE feature select macro; see Section 1.3.4 [Feature Test Macros], page 15.

_CS_LFS64_LDFLAGS The returned string specifies which additional flags must be given to the linker if a source is compiled using the _LARGEFILE64_SOURCE feature select macro; see Section 1.3.4 [Feature Test Macros], page 15.

_CS_LFS64_LIBS The returned string specifies which additional libraries must be linked to the application if a source is compiled using the _LARGEFILE64_SOURCE feature select macro; see Section 1.3.4 [Feature Test Macros], page 15.

_CS_LFS64_LINTFLAGS The returned string specifies which additional flags must be given to the lint tool if a source is compiled using the _LARGEFILE64_SOURCE feature select macro; see Section 1.3.4 [Feature Test Macros], page 15. The way to use confstr without any arbitrary limit on string size is to call it twice: first call it to get the length, allocate the buffer accordingly, and then call confstr again to fill the buffer, like this:

```
char *
get_default_path (void)
{
  size_t len = confstr (_CS_PATH, NULL, 0);
  char *buffer = (char *) xmalloc (len);
  if (conf str (_CS_PATH, buf, len + 1) == 0)
  {
     free (buffer);
     return NULL;
 }
 return buffer;
}
```

CHAPTER 6

Internal probes

In order to aid in debugging and monitoring internal behavior, the GNU C Library exposes nearly-zero-overhead SystemTap probes marked with the libc provider. These probes are not part of the GNU C Library stable ABI, and they are subject to change or removal across releases. Our only promise with regard to them is that, if we find a need to remove or modify the arguments of a probe, the modified probe will have a different name, so that program monitors relying on the old probe will not get unexpected arguments.

6.1 Memory Allocation Probes

These probes are designed to signal relatively unusual situations within the virtual memory subsystem of the GNU C Library.

memory_sbrk_more (void *$arg1, size t $arg2) [Probe]

This probe is triggered after the main arena is extended by calling sbrk. Argument $arg1 is the additional size requested to sbrk, and $arg2 is the pointer that marks the end of the sbrk area, returned in response to the request.

memory_sbrk_less (void *$arg1, size t $arg2) [Probe]

This probe is triggered after the size of the main arena is decreased by calling sbrk. Argument $arg1 is the size released by sbrk (the positive value, rather than the negative value passed to sbrk), and $arg2 is the pointer that marks the end of the sbrk area, returned in response to the request.

memory_heap_new (void *$arg1, size t $arg2) [Probe]

This probe is triggered after a new heap is mmaped. Argument $arg1 is a pointer to the base of the memory area, where the heap_info data structure is held, and $arg2 is the size of the heap.

memory_heap_free (void *$arg1, size t $arg2) [Probe]

This probe is triggered before (unlike the other sbrk and heap probes) a heap is completely removed via munmap. Argument $arg1 is a pointer to the heap, and $arg2 is the size of the heap.

memory_heap_more (void *$arg1, size t $arg2) [Probe]

This probe is triggered after a trailing portion of an mmaped heap is extended. Argument $arg1 is a pointer to the heap, and $arg2 is the new size of the heap.

memory_heap_less (void *$arg1, size t $arg2) [Probe]

This probe is triggered after a trailing portion of an mmaped heap is released. Argument $arg1 is a pointer to the heap, and $arg2 is the new size of the heap.

memory_malloc_retry (size t $arg1) [Probe]

memory_realloc_retry (size t $arg1, void *$arg2) [Probe] memory_memalign_retry (size t $arg1, size t $arg2) [Probe]

memory_calloc_retry (size t $arg1) [Probe]

These probes are triggered when the corresponding functions fail to obtain the requested amount of memory from the arena in use, before they call arena_get_retry to select an alternate arena in which to retry the allocation. Argument $arg1 is the amount of memory requested by the user; in the calloc case, that is the total size computed from both function arguments. In the realloc case, $arg2 is the pointer to the memory area being resized. In the memalign case, $arg2 is the alignment to be used for the request, which may be stricter than the value passed to the memalign function. A memalign probe is also used by functions posix_memalign, valloc and pvalloc.

Note that the argument order does not match that of the corresponding two-argument functions, so that in all of these probes the user-requested allocation size is in $arg1.

memory_arena_retry (size t $arg1, void *$arg2) [Probe]

This probe is triggered within arena_get_retry (the function called to select the alternate arena in which to retry an allocation that failed on the first attempt), before the selection of an alternate arena. This probe is redundant, but much easier to use when it's not important to determine which of the various memory allocation functions is failing to allocate on the first try. Argument $arg1 is the same as in the functionspecific probes, except for extra room for padding introduced by functions that have to ensure stricter alignment. Argument $arg2 is the arena in which allocation failed.

memory_arena_new (void *$arg1, size t $arg2) [Probe]

This probe is triggered when malloc allocates and initializes an additional arena (not the main arena), but before the arena is assigned to the running thread or inserted into the internal linked list of arenas. The arena's malloc_state internal data structure is located at $arg1, within a newly-allocated heap big enough to hold at least $arg2 bytes.

memory_arena_reuse (void *$arg1, void *$arg2) [Probe]

This probe is triggered when malloc has just selected an existing arena to reuse, and (temporarily) reserved it for exclusive use. Argument $arg1 is a pointer to the newly-selected arena, and $arg2 is a pointer to the arena previously used by that thread. This occurs within reused_arena, right after the mutex mentioned in probe **memory_** arena_reuse_wait is acquired; argument $arg1 will point to the same arena. In this configuration, this will usually only occur once per thread. The exception is when a thread first selected the main arena, but a subsequent allocation from it fails: then, and only then, may we switch to another arena to retry that allocations, and for further allocations within that thread.

memory_arena_reuse_wait (void *$arg1, void *$arg2, void *$arg3) [Probe]

This probe is triggered when malloc is about to wait for an arena to become available for reuse. Argument $arg1 holds a pointer to the mutex the thread is going to wait on, $arg2 is a pointer to a newly-chosen arena to be reused, and $arg3 is a pointer to the arena previously used by that thread. This occurs within reused_arena, when a thread first tries to allocate memory or needs a retry after a failure to allocate from the main arena, there isn't any free arena, the maximum number of arenas has been reached, and an existing arena was chosen for reuse, but its mutex could not be immediately acquired. The mutex in $arg1 is the mutex of the selected arena.

memory_arena_reuse_free_list (void *$arg1) [Probe]

This probe is triggered when malloc has chosen an arena that is in the free list for use by a thread, within the get_free_list function. The argument $arg1 holds a pointer to the selected arena.

memory_mallopt (int $arg1, int $arg2) [Probe]

This probe is triggered when function mallopt is called to change malloc internal configuration parameters, before any change to the parameters is made. The arguments $arg1 and $arg2 are the ones passed to the mallopt function.

memory_mallopt_mxfast (int $arg1, int $arg2) [Probe]

This probe is triggered shortly after the memory_mallopt probe, when the parameter to be changed is M_MXFAST, and the requested value is in an acceptable range. Argument $arg1 is the requested value, and $arg2 is the previous value of this malloc parameter.

memory_mallopt_trim_threshold (int $arg1, int $arg2, int $arg3) [Probe]

This probe is triggere shortly after the memory_mallopt probe, when the parameter to be changed is M_TRIM_THRESHOLD. Argument $arg1 is the requested value, $arg2 is the previous value of this malloc parameter, and $arg3 is nonzero if dynamic threshold adjustment was already disabled.

memory_mallopt_top_pad (int $arg1, int $arg2, int $arg3) [Probe]

This probe is triggered shortly after the memory_mallopt probe, when the parameter to be changed is M_TOP_PAD. Argument $arg1 is the requested value, $arg2 is the previous value of this malloc parameter, and $arg3 is nonzero if dynamic threshold adjustment was already disabled.

memory_mallopt_mmap_threshold (int $arg1, int $arg2, int $arg3) [Probe]

This probe is triggered shortly after the memory_mallopt probe, when the parameter to be changed is M_MMAP_THRESHOLD, and the requested value is in an acceptable range. Argument $arg1 is the requested value, $arg2 is the previous value of this malloc parameter, and $arg3 is nonzero if dynamic threshold adjustment was already disabled.

memory_mallopt_mmap_max (int $arg1, int $arg2, int $arg3) [Probe]

This probe is triggered shortly after the memory_mallopt probe, when the parameter to be changed is M_MMAP_MAX. Argument $arg1 is the requested value, $arg2 is the previous value of this malloc parameter, and $arg3 is nonzero if dynamic threshold adjustment was already disabled.

memory_mallopt_check_action (int $arg1, int $arg2) [Probe]

This probe is triggered shortly after the memory_mallopt probe, when the parameter to be changed is M_CHECK_ACTION. Argument $arg1 is the requested value, and $arg2 is the previous value of this malloc parameter.

memory_mallopt_perturb (int $arg1, int $arg2) [Probe]

This probe is triggered shortly after the memory_mallopt probe, when the parameter to be changed is M_PERTURB. Argument $arg1 is the requested value, and $arg2 is the previous value of this malloc parameter.

memory_mallopt_arena_test (int $arg1, int $arg2) [Probe]

This probe is triggered shortly after the memory_mallopt probe, when the parameter to be changed is M_ARENA_TEST, and the requested value is in an acceptable range. Argument $arg1 is the requested value, and $arg2 is the previous value of this malloc parameter.

memory_mallopt_arena_max (int $arg1, int $arg2) [Probe]

This probe is triggered shortly after the memory_mallopt probe, when the parameter to be changed is M_ARENA_MAX, and the requested value is in an acceptable range. Argument $arg1 is the requested value, and $arg2 is the previous value of this malloc parameter.

memory_mallopt_free_dyn_thresholds (int $arg1, int $arg2) [Probe]

This probe is triggered when function free decides to adjust the dynamic brk/mmap thresholds. Argument $arg1 and $arg2 are the adjusted mmap and trim thresholds, respectively.

6.2 Mathematical Function Probes

Some mathematical functions fall back to multiple precision arithmetic for some inputs to get last bit precision for their return values. This multiple precision fallback is much slower than the default algorithms and may have a significant impact on application performance. The systemtap probe markers described in this section may help you determine if your application calls mathematical functions with inputs that may result in multiple-precision arithmetic.

Unless explicitly mentioned otherwise, a precision of 1 implies 24 bits of precision in the mantissa of the multiple precision number. Hence, a precision level of 32 implies 768 bits of precision in the mantissa.

slowexp_p6 (double $arg1, double $arg2) [Probe]

This probe is triggered when the exp function is called with an input that results in multiple precision computation with precision 6. Argument $arg1 is the input value and $arg2 is the computed output.

slowexp_p32 (double $arg1, double $arg2) [Probe]

This probe is triggered when the exp function is called with an input that results in multiple precision computation with precision 32. Argument $arg1 is the input value and $arg2 is the computed output.

slowpow_p10 (double $arg1, double $arg2, double $arg3, double [Probe] $arg4)

This probe is triggered when the pow function is called with inputs that result in multiple precision computation with precision 10. Arguments $arg1 and $arg2 are the input values, $arg3 is the value computed in the fast phase of the algorithm and $arg4 is the final accurate value.

slowpow_p32 (double $arg1, double $arg2, double $arg3, double [Probe] $arg4)

This probe is triggered when the pow function is called with an input that results in multiple precision computation with precision 32. Arguments $arg1 and $arg2 are the input values, $arg3 is the value computed in the fast phase of the algorithm and $arg4 is the final accurate value.

slowlog (int $arg1, double $arg2, double $arg3) [Probe]

This probe is triggered when the log function is called with an input that results in multiple precision computation. Argument $arg1 is the precision with which the computation succeeded. Argument $arg2 is the input and $arg3 is the computed output.

slowlog_inexact (int $arg1, double $arg2, double $arg3) [Probe]

This probe is triggered when the log function is called with an input that results in multiple precision computation and none of the multiple precision computations result in an accurate result. Argument $arg1 is the maximum precision with which computations were performed. Argument $arg2 is the input and $arg3 is the computed output.

slowatan2 (int $arg1, double $arg2, double $arg3, double $arg4) [Probe]

This probe is triggered when the atan2 function is called with an input that results in multiple precision computation. Argument $arg1 is the precision with which computation succeeded. Arguments $arg2 and $arg3 are inputs to the atan2 function and $arg4 is the computed result.

slowatan2_inexact (int $arg1, double $arg2, double $arg3, double [Probe] $arg4)

This probe is triggered when the atan function is called with an input that results in multiple precision computation and none of the multiple precision computations result in an accurate result. Argument $arg1 is the maximum precision with which computations were performed. Arguments $arg2 and $arg3 are inputs to the atan2 function and $arg4 is the computed result.

slowatan (int $arg1, double $arg2, double $arg3) [Probe]

This probe is triggered when the atan function is called with an input that results in multiple precision computation. Argument $arg1 is the precision with which computation succeeded. Argument $arg2 is the input to the atan function and $arg3 is the computed result.

slowatan_inexact (int $arg1, double $arg2, double $arg3) [Probe]

This probe is triggered when the atan function is called with an input that results in multiple precision computation and none of the multiple precision computations result in an accurate result. Argument $arg1 is the maximum precision with which computations were performed. Argument $arg2 is the input to the atan function and $arg3 is the computed result.

slowtan (double $arg1, double $arg2) [Probe]

This probe is triggered when the tan function is called with an input that results in multiple precision computation with precision 32. Argument $arg1 is the input to the function and $arg2 is the computed result.

slowasin (double $arg1, double $arg2) [Probe]

This probe is triggered when the asin function is called with an input that results in multiple precision computation with precision 32. Argument $arg1 is the input to the function and $arg2 is the computed result.

slowacos (double $arg1, double $arg2) [Probe]

This probe is triggered when the acos function is called with an input that results in multiple precision computation with precision 32. Argument $arg1 is the input to the function and $arg2 is the computed result.

slowsin (double $arg1, double $arg2) [Probe]

This probe is triggered when the sin function is called with an input that results in multiple precision computation with precision 32. Argument $arg1 is the input to the function and $arg2 is the computed result.

slowcos (double $arg1, double $arg2) [Probe]

This probe is triggered when the cos function is called with an input that results in multiple precision computation with precision 32. Argument $arg1 is the input to the function and $arg2 is the computed result.

slowsin_dx (double $arg1, double $arg2, double $arg3) [Probe]

This probe is triggered when the sin function is called with an input that results in multiple precision computation with precision 32. Argument $arg1 is the input to the function, $arg2 is the error bound of $arg1 and $arg3 is the computed result.

slowcos_dx (double $arg1, double $arg2, double $arg3) [Probe]

This probe is triggered when the cos function is called with an input that results in multiple precision computation with precision 32. Argument $arg1 is the input to the function, $arg2 is the error bound of $arg1 and $arg3 is the computed result.

6.3 Non-local Goto Probes

These probes are used to signal calls to setjmp, sigsetjmp, longjmp or siglongjmp.

setjmp (void *$arg1, int $arg2, void *$arg3) [Probe]

This probe is triggered whenever setjmp or sigsetjmp is called. Argument $arg1 is a pointer to the jmp_buf passed as the first argument of setjmp or sigsetjmp, $arg2 is the second argument of sigsetjmp or zero if this is a call to setjmp and $arg3 is a pointer to the return address that will be stored in the jmp_buf.

longjmp (void *$arg1, int $arg2, void *$arg3) [Probe]

This probe is triggered whenever longjmp or siglongjmp is called. Argument $arg1 is a pointer to the jmp_buf passed as the first argument of longjmp or siglongjmp, $arg2 is the return value passed as the second argument of longjmp or siglongjmp and $arg3 is a pointer to the return address longjmp or siglongjmp will return to. The longjmp probe is triggered at a point where the registers have not yet been restored to the values in the jmp_buf and unwinding will show a call stack including the caller of longjmp or siglongjmp.

longjmp_target (void *$arg1, int $arg2, void *$arg3) [Probe]

This probe is triggered under the same conditions and with the same arguments as the longjmp probe. The longjmp_target probe is triggered at a point where the registers have been restored to the values in the jmp_buf and unwinding will show a call stack including the caller of setjmp or sigsetjmp.

Debugging support

Applications are usually debugged using dedicated debugger programs. But sometimes this is not possible and, in any case, it is useful to provide the developer with as much information as possible at the time the problems are experienced. For this reason a few functions are provided which a program can use to help the developer more easily locate the problem.

7.1 Backtraces

A *backtrace* is a list of the function calls that are currently active in a thread. The usual way to inspect a backtrace of a program is to use an external debugger such as gdb. However, sometimes it is useful to obtain a backtrace programmatically from within a program, e.g., for the purposes of logging or diagnostics.

The header file *execinfo.h* declares three functions that obtain and manipulate back- traces of the current thread.

int backtrace (void **buffer, int size) [Function] Preliminary: | MT-Safe | AS-Unsafe init heap dlopen plugin lock | AC-Unsafe init mem lock fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The *backtrace* function obtains a backtrace for the current thread, as a list of point- ers, and places the information into buffer. The argument size should be the number of void * elements that will fit into buffer. The return value is the actual number of entries of *buffer* that are obtained, and is at most size.

The pointers placed in *buffer* are actually return addresses obtained by inspecting the stack, one return address per stack frame.

Note that certain compiler optimizations may interfere with obtaining a valid back- trace. Function inlining causes the inlined function to not have a stack frame; tail call optimization replaces one stack frame with another; frame pointer elimination will stop *backtrace* from interpreting the stack contents correctly.

char * **backtrace_symbols (void** ***const *buffer, int size) [Function]*** Preliminary: | MT-Safe | AS-Unsafe heap | AC-Unsafe mem lock | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The *backtrace_symbols* function translates the information obtained from the backtrace function into an array of strings. The argument buffer should be a pointer to an array of addresses obtained via the *backtrace* function, and size is the number of entries in that array (the return value of *backtrace*).

The return value is a pointer to an array of strings, which has size entries just like the array *buffer*. Each string contains a printable representation of the corresponding element of *buffer*. It includes the function name (if this can be determined), an offset into the function, and the actual return address (in hexadecimal).

Currently, the function name and offset only be obtained on systems that use the ELF binary format for programs and libraries. On other systems, only the hexadecimal return address will be present. Also, you may need to pass additional flags to the linker to make the function names available to the program. (For example, on systems using GNU ld, you must pass (**-rdynamic**.)

The return value of *backtrace_symbols* is a pointer obtained via the *malloc* function, and it is the responsibility of the caller to *free* that pointer. Note that only the return value need be freed, not the individual strings. The return value is *NULL* if sufficient memory for the strings cannot be obtained.

void backtrace_symbols_fd (void *const *buffer, int size, int fd) [Function] Preliminary: | MT-Safe | AS-Safe | AC-Unsafe lock | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The *backtrace_symbols_fd* function performs the same translation as the function *backtrace_symbols* function. Instead of returning the strings to the caller, it writes the strings to the file descriptor fd, one per line. It does not use the malloc function, and can therefore be used in situations where that function might fail.

The following program illustrates the use of these functions. Note that the array to contain the return addresses returned by *backtrace* is allocated on the stack. Therefore code like this can be used in situations where the memory handling via malloc does not work anymore (in which case the *backtrace_symbols* has to be replaced by a *backtrace_symbols_fd* call as well). The number of return addresses is normally not very large. Even complicated programs rather seldom have a nesting level of more than, say, 50 and with 200 possible entries probably all programs should be covered.

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>

/* Obtain a backtrace and print it to stdout. */
void
print_trace (void)
{
 void *array[10];
 size_t size;
 char **strings;
 size_t i;

 size = backtrace (array, 10);
 strings = backtrace_symbols (array, size);

 printf ("Obtained %zd stack frames.\n", size);

 for (i = 0; i < size; i++)
         printf ("%s\n", strings[i]);

 free (strings);
}

/* A dummy function to make the backtrace more interesting. */
void
dummy_function (void)
{
 print_trace ();
}

int
main (void)
{
 dummy_function ();
 return 0;
}
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search