System V Interface Definition, Fourth Edition
Volume 1

Copyright© 1983, 1984, 1985, 1986,1987, 1988, 1995 Novell, Inc.
All Rights Reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of the publisher.

Novell, Inc.
122 East 1700 South
Provo, UT 84606
U.S.A.

IMPORTANT NOTE TO USERS

While every effort has been made to ensure the accuracy of all information in this document, Novell assumes no liability to any party for any loss of damage caused by errors or omissions or by statements of any kind in the *System V Interface Definition*, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Novell further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Novell disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose.

Novell makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

Novell reserves the right to make changes without further notice to any products herein to improve reliability, function, or design.

TRADEMARKS

Ann Arbor is a trademark of Ann Arbor Terminals, Inc.
Beehive is a trademark of Beehive International.
Concept is a trademark of Human Designed Systems, Inc.
HP is a trademark of Hewlett–Packard Co.
LSI is a trademark of Lear Siegler, Inc.
Micro–Term, ACT and MIME are trademarks of Micro–Term, Inc.
OSF/Motif is a trademark of the Open Software Foundation
PostScript is a trademark of Adobe Systems.
Tektronix and Tektronix 4010 are registered trademarks of Tektronix, Inc.
TeleVideo is a registered trademark of TeleVideo Systems, Inc.
Teleray is a trademark of Research, Inc.
Teletype is a registered trademark of AT&T.
The X Window System is a trademark of MIT.
UNIX is a registered trademark in the USA and other countries, licensed exclusively through X/Open Company, Ltd.
VT100 is a trademark of Digital Equipment Corporation.
X/Open is a trademark of X/Open Company Limited.

Volume 1 Table of Contents

Preface

1 GENERAL INTRODUCTION

2 BASE SYSTEM INTRODUCTION

3 BASE SYSTEM DEFINITIONS

4 BASE SYSTEM ENVIRONMENT ROUTINES

5 BASE OS SERVICE ROUTINES

6 BASE OS LIBRARY ROUTINES

7 BASE SYSTEM DEVICES INTRODUCTION

8 BASE SYSTEM DEVICES

9 KERNEL EXTENSION INTRODUCTION

10 KERNEL EXTENSION ENVIRONMENT ROUTINES

11 KERNEL EXTENSION OS SERVICE ROUTINES

12 MULTITHREADING EXTENSION INTRODUCTION

13 MULTITHREADING EXTENSION OS SERVICE ROUTINES

14 MULTITHREADING EXTENSION LIBRARY ROUTINES

PREFACE

The *System V Interface Definition* (SVID) specifies an operating system environment that allows users to create applications software that is independent of any particular computer hardware. The *System V Interface Definition* applies to computers that range from personal computers to mainframes. Applications that conform to this specification will allow users to take advantage of changes in technology and to choose the computer that best meets their needs from among many manufacturers while retaining a common computing environment.

The *System V Interface Definition* specifies the operating system components available to both end-users and application programs. It defines the functionality of components, but not the implementation. The *System V Interface Definition* specifies the source code interfaces of each operating system component, as well as the run-time behavior seen by an application program or an end-user. The emphasis is on defining a common computing environment for application programs and end-users, not on the internals of the operating system, such as the scheduler or memory manager.

An application program using only components defined in the *System V Interface Definition* will be compatible with, and portable to, any computer that supports the System V Interface. While the source code may have to be re-compiled to move an application program to a new computer system that supports the System V Interface, the presence and behavior of the operating system components as defined by the *System V Interface Definition* would be assured.

The *System V Interface Definition* is organized into a Base System Definition plus a series of Extension Definitions. The Base System Definition specifies the components that all System V operating systems must provide. The Extensions to the Base System are not required to be present in a System V operating system, but when a component is present, it must conform to the specified functionality. The *System V Interface Definition* allows end-users and application developers to identify the features and functions available to them on any System V operating system.

The *System V Interface Definition* is compliant with POSIX 1003.1-1990 Full Use Standard, X3.159-1992 (ANSI C), ISO / IEC 9899-1992 (ISO C), X / Open Portability Guide Isse 4 (XPG4) System Interfaces and Headers (XSH4), and will continue to evolve towards compliance with other industry standards as they are approved.

General Introduction

Audience and Purpose

The *System V Interface Definition* (SVID) is intended for use by anyone who must understand the operating system components that are consistent across all System V environments. As such, its primary audience is the application developer who is building **C** language application programs having source code that must be portable from one System V environment to another. A system builder should also view these volumes as necessary tools for supporting a System V environment that will host such applications.

This publication is intended to fulfill the following major purposes:

- To serve as a single reference source for the definition of the external interfaces to services that are provided by all System V environments. These services are designated as the Base System. This includes source-code interfaces and run-time behavior as seen by an application program. It does not include the details of how the operating system implements these functions.
- To define all additional services (such as graphics, networking and data management) at an equivalent external interface level and to group these services into Extensions to the Base System.
- To serve as a complete definition of System V external interfaces, so that application source code that conforms to these interfaces and is compiled in an environment that conforms to these interfaces, will execute as defined in a System V environment. It is assumed that source code is recompiled for the proper target hardware. The basic objective of this document is to facilitate the writing of application program source code that is directly portable across all System V implementations. Facilities outside the Base System would require installation of the appropriate Extension on the target environment.

Structure and Content

Partitioning into Base System and Extensions

The *System V Interface Definition* partitions System V components into a Base System and the Extensions to that Base System. This does not change the definition of System V. Instead, the approach recognizes that the entire functionality of System V may be unnecessary in certain environments, especially on small hardware configurations. It also recognizes that different computing environments require some functions that others do not.

The Base System functionality has been structured to provide a minimal, standalone run-time environment for application programs originally written in a high-level language, such as **C**. In this environment, the end-user is not expected to interact directly with the traditional System V shell and commands. An example of such a system would be a dedicated-use system, that is, one devoted to a single application, such as a vertically integrated application package for managing a legal office. To execute, many applications programs require only the components in the Base System; other applications require one or more Extensions.

The Extensions to this Base System have been structured to provide a growth path, in natural functional increments, that leads to a full System V configuration, and to provide a mechanism for the introduction of new technology. The division between the Base System and the Extensions allows system builders to create machines, tailored for different purposes and markets, in an orderly fashion. Thus, a small business / professional computer system designed for novice single-users might include only the Base System and the Basic Utilities Extension. A system for advanced business / professional users might add the Advanced Utilities Extension to this. A system designed for high-level language software development would include the Base System, the Kernel Extension, and the Basic Utilities, Advanced Utilities, and Software Development Extensions. Although the Extensions are not meant to specify the physical packaging of System V for a particular product, it is expected that the Extensions will lead to a fairly consistent packaging scheme.

This partitioning allows an application to be built using a basic set of components that are consistent across all System V implementations. This basic set is the Base System. Where necessary, an application developer can choose to use components from an Extension and require the run-time environment to support that Extension in addition to the Base System.

Facilities or side effects that are not explicitly stated in the SVID are not guaranteed, and should not be used by applications that require portability.

Conforming Systems

All conforming systems must support the source-code interfaces and runtime behavior of all the components of the Base System. A system may conform to none or some Extensions. All the components of an Extension must be present for a system to meet the requirements of the Extension. This does not preclude a system from including only a few components from some Extension, but the system would *not* then be said to have the Extension. Some Extensions require that other Extensions be present on a system. For example, the Advanced Utilities Extension requires the Basic Utilities Extension. In rare instances particular routines are explicitly marked in the SVID as optional and may not be present on all conforming systems.

An implementation of System V may conform to earlier issues of the SVID.

Organization of Technical Information

SVID, Fourth Edition (SVID 4) is composed of Volumes 1 through 4. The volumes are organized as follows:

Volume 1	Base System Kernel Extension Multithreading Extension
Volume 2	Basic Utilities Extension Advanced Utilities Extension Administered Systems Extension
Volume 3	Programming Language Specification Software Development Extension Terminal Interface Extension Real Time and Memory Management Extension Remote Services Extension Window System Extension Enhanced Security Extension Auditing Extension Remote Administration Extension

The SVID defines the source-code interface and the run-time behavior of the components that constitute the Base System and each Extension. Components include, for example, operating system service routines, general library routines, system data files, special device files, and end-user utilities (commands).

When referred to individually, components are identified by a suffix of the form (*XX_YYY*) where *XX* identifies the Base System or the Extension containing the component and *YYY* identifies the type of the component. For example, components defined in the Operating System Service Routines section of the Base System are identified by (BA_OS), components defined in General Library Routines

of the Base System are identified by (BA_LIB), and components defined in the Operating System Service Routines section of the Kernel Extension are identified by (KE_OS).

The definition of the Base System includes an introduction, followed by chapters that provide detailed definitions of each component in the Base System. Similarly, the definition of each Extension includes an introduction, followed by chapters that provide detailed definitions of each component in the Extension.

Pages containing the detailed component definitions are labeled with the name of the component being defined. Some utilities and routines are described with other related utilities or routines and, therefore, do not have detailed definition pages of their own.

Each component definition follows the same structure. The sections are listed below; not all the following sections may be present in each description. Sections entitled **EXAMPLE** and **USAGE** are not considered part of the formal definition of a component.

- **NAME** — name of component
- **SYNOPSIS** — summary of source code or user-level interface
- **DESCRIPTION** — interface and run-time behavior
- **RETURN VALUE** — value returned by the function
- **ERRORS** — possible error conditions
- **FILES** — names of files used
- **USAGE** — guidance on use
- **EXAMPLE** — example
- **SEE ALSO** — list of related components
- **Future Directions** — planned enhancements
- **LEVEL** — see **Mechanism For Evolution** below

In general, components that are utilities do not have a **RETURN VALUE** section. Except as noted in the detailed definition for a particular utility, utilities return a zero exit code for *success*, and non-zero for *failure*.

The component definitions are similar in format to AT&T System V manual pages, but have been extended or modified as follows:

- Function prototype format has been used as the *presentation format* in the **SYNOPSIS** for SVID 4. The consistent use of function prototypes is intended to provide an easy to use interface to users of the SVID and is not required for conformance.

- All machine-specific information has been removed. All implementation-specific constants have been replaced by symbolic names, which are defined in a separate section. The symbolic names correspond to those defined by the the IEEE 1003.1-1990 Standard to be in a **<limits.h>** header file; however, in this document, they are *not* meant to be read as symbolic constants defined in header files. For maximum portability, applications should not depend upon any particular behavior that is implementation-defined.
- A section entitled **USAGE** has been added to guide application developers in the expected or recommended usage of certain components. Operating system services and library routines are used only by programs, but utilities may be used by programs, end-users or administrators. The **USAGE** paragraph indicates which of these three is appropriate for a particular utility (this is not meant to be prescriptive, but rather to give guidance). The following terms are used in the **USAGE** paragraph: *application program*, *end-user*, *administrator*, or *general*. The term *general* indicates that the utility may be used by all three: application programs, end-users and administrators.
- A section entitled **Future Directions** has been added to selected component definitions. This section indicates the way in which a component will evolve. The information ranges from specific changes in functionality to more general indications of proposed development.
- A section entitled **LEVEL** defines the commitment level of each component.

  Level 1 components will remain in the SVID and can be modified only in upwardly compatible ways. Any change in the definition of the component will preserve the previous source-code interface and run-time behavior to ensure that the component remains upwardly compatible. A Level 1 component may however contain some features that are defined as Level 2. This occurs in cases in which a portion of a component is evolving in a non-upwardly compatible way, but the basic functionality of the component remains unchanged.

  Level 2 components will remain unchanged for at least three years following entry into Level 2, after which time the component may be modified in a non-upwardly compatible way or may be dropped from the SVID. This mechanism also applies to Level 2 portions of a Level 1 component. Level 2 components are labeled with the starting date of this three-year period.

Mechanism For Evolution

The SVID will be reissued as necessary to reflect developments in the System V Interface. In conjunction with these updates, the following changes may be made to the definitions:

- Level 1 components may be moved to Level 2. The date of their entry into Level 2 will be the date of the reissue of the SVID in which the change is made.
- In cases in which a published Industry Standard has specified behavior that is not upwardly compatible with the behavior documented in the SVID for a Level 1 component, the component will change to reflect the behavior specified by the standard. Wherever possible both the behavior defined by the Industry Standard and the behavior documented in the SVID will be supported. The behavior documented in the SVID will be preserved for the Level 2 migration period.
- Components may move from existing Extensions into the Base System. Components will not move from the Base System into an Extension.
- New Extensions may be introduced with completely new functionality.
- Notification of changes to SVID components may be done as required to facilitate conformance to industry standards. This will allow customers a more orderly migration to the standard.

Evolution Toward Industry Standards

Novell is committed to compliance with standards published by IEEE, ANSI, ISO, X/Open and other major standards bodies. Where conformance to an industry standard causes an incompatibility with SVID, the incompatible component, or the incompatible feature of the component will move to Level 2 (see **Mechanism For Evolution**). The **Future Directions** section for the affected component will describe how the component will change in the future. In this case, compliance to the current SVID behavior or the new industry standard behavior will satisfy SVID compliance. The incompatible component, or component feature will be indicated by a (‡).

C Language Definition

Source code interfaces described in the SVID are for the **C** language.

Major Features

The content changes in the SVID, 4th Edition, reflect the major feature changes in UNIX System V, namely:

- Multiprocessing
- Dynamically Loadable Modules (DLM)
- Internationalization Enhancements and Standards conformance
  - conformance to the ISO C Multibyte Support Extensions,
  - conformance to XPG4 Systems Interfaces, Headers, and Compilation System components,
  - conformance to portions of the NCEG extension to the **math** and systems libraries,
  - more extensive POSIX .2 functionality,
  - support for the XPG4 Transport Interface Specification.
- Graphics

Future Directions

The following describes some areas in the SVID where changes or evolution are expected. Refer also to the **Future Directions** sections that appear in the SVID manual page descriptions.

Internationalization

The SVID, 4th Edition, reflects the support provided in UNIX System V, in support of the ISO C Multibyte Support Extension (**ISO C MSE**) for wide-character and multibyte-character handling; as well provision of the XPG4 Worldwide Portability Interfaces, required for XPG4 conformance. As in earlier releases, more System V commands have been modified to use internationalized messaging and localization facilities.

In the future, support for the POSIX 1003.2 enhanced regular expression handling will be provided,as well as further internationalization of commands and utilities.

Pthreads

POSIX 1003.1c Threads ("**Pthreads**) are not yet finalized. In the SVID, 4th Edition, UNIX System V threads are represented. UNIX System V threads interfaces offer greater functionality than **Pthreads** provides, and will thus fill application needs for a considerable period of time after the initial standardization of **Pthreads**.

The UNIX System V threads will be given the fullest support for compatibility and migration standard that is granted to other Level 1 interfaces in the SVID. The future evolution of these interfaces, where known, will be noted in the appropriate sections of the SVID 4th Edition.

Real Time

Novell is committed to support the standardization of a Real Time interface as defined by POSIX. Full conformance to this standard will be considered in the future.

Security

Novell is working in conjunction with the POSIX P1003.6 security working group in developing an IEEE security standard. Full conformance to the IEEE standard will be strongly considered after its formal approval.

Asynchronous I/O

This version of the SVID includes the asynchronous I/O interfaces that are in full conformance to the POSIX 1003.1c interfaces.

DCE

DCE and Systems Management functionality may be included in the SVID in the future.

Base System Introduction

The Base System supports a minimal run-time environment for executable applications. The Base System defines a basic set of System V components needed by applications programs. This basic set would be supported by any conforming system. It defines each component's source-code interface and run-time behavior, but does not specify its implementation. Source code interfaces described are for the **C** language. While only the run-time behavior of these components is supported by the Base System, the source-code interfaces to these components are defined because an objective of the SVID is to facilitate application program source-code portability across all System V implementations. It is assumed that an application program targeted to run on a system that provides only the Base System (a run-time environment) would be *compiled* on a system supporting software development.

No end-user level utilities (commands) are defined in the Base System. Executable application programs designed for maximum portability are expected to use library routines rather than System V end-user level utilities. For example, an application program written in **C** would use the **chmod()** routine to change the owner of a file rather than using the **chmod** user-level utility. This does not say that an application program running in a target environment that supports only the Base System cannot execute another program. Using the **system** routine, an application can execute another program or application.

It should be noted that some Extensions may add features to components defined in the Base System. Additional features that are supported in an extended environment are described with the Extension in a section titled **effects(***xx***_ENV)**. [See, for example, **effects(KE_ENV)**.]

OS Service Routines

The Base OS Service routines provide access to and control over system resources such as memory, files and process execution. Some System V routines that provide operating system services are not supported by the Base System. An application-program that uses any of these would require an *extended* environment. [See, for example, the Kernel Extension Definition.]

There are three groups of Base OS Service Routines (listed below), which reflect recommended usage by application programs.

Group 1 should fulfill the needs of most application programs.

Group 2 should be used by application programs only when some special need requires it. For example, application programs, when possible, should use the routine **system()** rather than the routines **fork()** and exec because it is easier to use and supplies more functionality. The corresponding Standard Input/Output, *stdio* routines [see "stdio routines" in the *Base System Definitions* chapter] should be used instead of the routines **close()**, **creat()**, **lseek()**, **open()**, **read()** and **write()** (for example, the *stdio* routine **fopen()** should be used rather than the routine **open()**).

Group 3 routines, although defined as part of the basic set of routines supported by any System V operating system, are not expected to be used by application programs. These routines are used by other components of the Base System.

The following OS service routines are supported by a SVID-compliant Base system. Items marked with a star (*) are Level 2, as defined in the *General Introduction* to this volume. Items marked with a dagger (†) are new to this issue of the SVID.

Base OS Service Routines (group 1)

abort	**fchown**	**getcontext**	**malloc**	**rewinddir**
access	**fclose**	**getcwd**	**mallopt***	**rmdir**
adjtime*	**fcntl**	**getegid**	**mkdir**	**seekdir**
alarm	**fdopen**	**geteuid**	**mkfifo**	**setcontext**
atexit	**feof**	**getgid**	**mknod**	**setgid**
calloc	**ferror**	**getgroups**	**opendir**	**setgroups**
cfgetispeed	**fflush**	**getmsg**	**pathconf**	**setlocale**
cfgetospeed	**fgetpos**	**getpgid**	**pause**	**setpgid**
cfsetispeed	**fileno**	**getpgrp**	**pclose**	**setrlimit**
cfsetospeed	**filepriv**	**getpid**	**pipe**	**setsid**
chdir	**fopen**	**getpmsg**	**poll**	**setuid**
chmod	**fpathconf**	**getppid**	**popen**	**sigaction**
chown	**fread**	**getrlimit**	**procpriv**	**sigaddset**
clearerr	**free**	**getsid**	**putmsg**	**sigaltstack**
closedir	**freopen**	**getuid**	**putpmsg**	**sigdelset**
confstr†	**fseek**	**ioctl**	**raise**	**sigemptyset**
cuserid	**fsetpos**	**kill**	**readdir**	**sigfillset**
dup	**fstat**	**lchown**	**readlink**	**sigismember**
dup2	**fstatvfs**	**link**	**realloc**	**signal**
exit	**fsync**	**lockf**	**remove**	**sigpending**
fchdir	**ftell**	**lstat**	**rename**	**sigprocmask**
fchmod	**fwrite**	**mallinfo***	**rewind**	**sigsend**

Base OS Service Routines (group 1)

sigsendset	**stime**	**tcflush**	**tcsetpgrp**	**uname**
sigsuspend	**symlink**	**tcgetattr**	**telldir**	**unlink**
sigwait†	**sysconf**	**tcgetpgrp**	**time**	**utime**
sleep	**system**	**tcgetsid**	**times**	**wait**
stat	**tcdrain**	**tcsendbreak**	**ulimit**	**waitid**
statvfs	**tcflow**	**tcsetattr**	**umask**	**waitpid**

Base OS Service Routines (group 2)

close	**dlsym†**	**execv**	**lseek**	**readv**
creat	**execl**	**execve**	**mount**	**umount**
dlclose†	**execle**	**execvp**	**open**	**write**
dlerror†	**execlp**	**fork**	**read**	**writev**
dlopen†				

Base OS Service Routines (group 3)

_exit	**sync**

Library Routines

The Base System library routines perform a wide range of useful tasks, including

- mathematical functions
- string and character handling, including XPG4 Worldwide Portability Interfaces and functions in the ISO C Multibyte Support Extension (MSE).
- networking functions
- general library functions (including I/O, searching and sorting routines)

The *run-time* behavior of these routines, as defined in the SVID, must be supported by any System V operating system. The libraries themselves are not required to be present on a system that consists only of the Base System. While the Base System is required to support the execution of application programs that use these routines, the Software Development Extension is required to support the *compilation* of those application programs.

The following routines are supported by the Base System (*exception:* items marked with a sharp (#) are optional and may not be present on all conforming systems). Items marked with a star (*) are Level 2, as defined in the *General Introduction* to this volume. Items marked with a dagger (†) are new to this issue of the SVID.

Mathematical Functions

abs	**ceil**	**fmod**	**ldiv**	**scalb**
acos	**cos**	**frexp**	**lgamma**	**sin**
acosh	**cosh**	**gamma***	**log**	**sinh**
asin	**div**	**hypot**	**log10**	**sqrt**
asinh	**erf**	**j0**	**logb**	**tan**
atan	**erfc**	**j1**	**modf**	**tanh**
atan2	**exp**	**jn**	**nextafter**	**y0**
atanh	**fabs**	**labs**	**pow**	**yn**
cbrt	**floor**	**ldexp**	**remainder**	

String and Character Handling

_tolower	**isgraph**	**memchr**	**strstr**	**wcscmp†**
_toupper	**islower**	**memcmp**	**strtod**	**wcscoll†**
advance*	**isnan**	**memcpy**	**strtof†**	**wcscpy†**
asctime	**isprint**	**memmove**	**strtok**	**wcscspn†**
atof	**ispunct**	**memset**	**strtol**	**wcsftime†**
atoi	**isspace**	**mktime**	**strtold†**	**wcslen†**
atol	**isupper**	**putwc†**	**strtoul**	**wcsncat†**
compile*	**iswalnum†**	**putwchar†**	**strxfrm**	**wcsncmp†**
crypt#	**iswalpha†**	**setkey#**	**swprintf†**	**wcsncpy†**
ctime	**iswcntrl†**	**snprintf†**	**swscanf†**	**wcspbrk†**
difftime	**iswctype†**	**step***	**toascii**	**wcsrchr†**
encrypt#	**iswdigit†**	**strcat**	**tolower**	**wcsrtombs†**
fgetwc†	**iswgraph†**	**strchr**	**toupper**	**wcsspn†**
fgetws†	**iswlower†**	**strcmp**	**towlower†**	**wcsstr†**
fputwc†	**iswprint†**	**strcoll**	**towupper†**	**wcstod†**
fputws†	**iswpunct†**	**strcpy**	**tzset**	**wcstof†**
ftok*	**iswspace†**	**strcspn**	**ungetwc†**	**wcstok†**
fwprintf†	**iswupper†**	**strdup**	**vfscanf†**	**wcstold†**
fwscanf†	**iswxdigit†**	**strerror**	**vfwprintf†**	**wcstombs**
getwc†	**isxdigit**	**strfmon†**	**vfwscanf†**	**wcstoul†**
getwchar†	**localeconv**	**strftime**	**vscanf†**	**wcswcs*†**
gmtime	**localtime**	**strlen**	**vsnprintf†**	**wcswidth†**
iconv_close†	**mblen**	**strlist†**	**vsscanf†**	**wcsxfrm†**
iconv_open†	**mbrlen†**	**strncat**	**vswprintf†**	**wctob†**
isalnum	**mbrtowc†**	**strncmp**	**vswscanf†**	**wctomb**
isalpha	**mbsinit†**	**strncpy**	**vwprintf†**	**wctype†**
isascii	**mbsrtowcs†**	**strpbrk**	**vwscanf†**	**wcwidth†**
isatty	**mbstowcs**	**strptime†**	**wcrtomb†**	**wprintf†**
iscntrl	**mbtowc**	**strrchr**	**wcscat†**	**wscanf†**
isdigit	**memccpy**	**strspn**	**wcschr†**	

Networking Functions

get_t_errno†	**t_connect**	**t_listen**	**t_rcvdis**	**t_sndrel**
set_t_errno†	**t_error**	**t_look**	**t_rcvrel**	**t_sndudata**
t_accept	**t_free**	**t_open**	**t_rcvudata**	**t_strerror†**
t_alloc	**t_getinfo**	**t_optmgmt**	**t_rcvuderr**	**t_sync**
t_bind	**t_getprotaddr†**	**t_rcv**	**t_snd**	**t_unbind**
t_close	**t_getstate**	**t_rcvconnect**	**t_snddis**	

addsev*	**fscanf**	**hdestroy**	**putenv**	**srand***
assert	**ftok***	**hsearch**	**puts**	**sscanf**
bsearch	**ftw**	**initgroups**	**putw**	**stdio**
catclose	**getc**	**jrand48***	**qsort**	**swab**
catgets	**getchar**	**lcong48***	**rand**	**swapcontext**
catopen	**getdate***	**lfind**	**regcomp†**	**tdelete**
clock	**getenv**	**lfmt***	**regerror†**	**tempnam**
ctermid	**getgrent**	**longjmp**	**regexec†**	**tfind**
drand48*	**getgrgid**	**lrand48***	**regfree†**	**tmpfile**
endgrent	**getgrnam**	**lsearch**	**scanf**	**tmpnam**
endpwent	**getlogin**	**makecontext**	**seed48***	**tsearch**
erand48*	**getopt**	**mktemp**	**setbuf**	**ttyname**
fattach	**getpwent**	**mrand48***	**setcat***	**twalk**
fdetach	**getpwnam**	**nftw**	**setgrent**	**ungetc**
fgetc	**getpwuid**	**nl_langinfo**	**setjmp**	**unlockpt**
fgetgrent	**gets**	**nrand48***	**setlabel***	**vfprintf†**
fgetpwent	**getsubopt**	**perror***	**setpwent**	**vlfmt***
fgets	**gettxt**	**pfmt***	**setvbuf**	**vpfmt***
fmtmsg*	**getw**	**printf**	**siglongjmp**	**vprintf**
fnmatch†	**glob†**	**procprivl**	**sigsetjmp**	**vsprintf**
fprintf	**globfree†**	**ptsname**	**sprintf**	**wordexp†**
fputc	**grantpt**	**putc**	**srand48***	**wordfree†**
fputs	**hcreate**	**putchar**		

Organization of Technical Information

The ‘‘Base OS Service Routines’’ chapter provides manual page descriptions of operating system service routines supported by this extension.

The ‘‘Base OS Library Routines’’ chapter provides manual page descriptions of general purpose library routines supported by this extension.

Base System Definitions

Active Transport User

A transport user that initiates a transport connection.

Appropriate Privileges

An implementation-defined means of associating privileges with a process with regard to functions that need special privileges. There may be zero or more such means.

ASCII Character Set

Maps of the ASCII character set, giving octal and hexadecimal equivalents of each character, appear below. Although the ASCII code does not use the eighth-bit in an octet, this bit must not be used for other purposes because codes for other languages may need to use it (see the section on *Internationalization* in the *General Introduction* to this volume.)

Octal map of ASCII character set.

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	ht	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042	"	043	#	044	$	045	%	046	&	047	’
050	(	051	)	052	*	053	+	054	,	055	–	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	‘	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	152	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w

170 x 171 y 172 z 173 { 174 | 175 } 176 ~ 177 del

FINAL COPY
June 15, 1995
File: ba_def.txt
svid

Hexadecimal map of ASCII character set.

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0a nl	0b vt	0c np	0d cr	0e so	0f si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 $	25 %	26 &	27 ’
28 (	29)	2a *	2b +	2c ,	2d –	2e .	2f /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3a :	3b ;	3c <	3d =	3e >	3f ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4a J	4b K	4c L	4d M	4e N	4f O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5a Z	5b [	5c \	5d]	5e ^	5f _
60 ‘	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6a j	6b k	6c l	6d m	6e n	6f o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7a z	7b {	7c |	7d }	7e ~	7f del

Asynchronous Execution

The mode of execution in which transport service functions do not wait for specific events to occur before returning control to the user, but instead return immediately if the event is not pending.

Background Process Group

A background process group is any process group that is a member of a session which has established a connection with a controlling terminal that is not in the foreground process group.

Connection Mode

A connection mode is a mode of transfer in which data is passed from one process to another over an established connection in a reliable, sequenced fashion. The connection may also be called a virtual circuit.

Connectionless (datagram) Mode

A connectionless (datagram) mode is a mode of transfer in which data is passed from one process to another in self-contained units (datagrams) with no logical relationship required among multiple units.

Controlling Process

A controlling process is a session leader that establishes a connection to a controlling terminal. Should the terminal subsequently cease to be a controlling terminal for the session leader's session, the session leader shall cease to be a controlling process.

Controlling Terminal

A controlling terminal is a terminal that is associated with one session. Each session may have at most one controlling terminal associated with it and vice versa. Certain input sequences from the controlling terminal cause signals to be sent to processes associated with the controlling terminal.

Directory

Directories organize files into a hierarchical system where directories are the nodes in the hierarchy. A directory is a file that catalogues the list of files, including directories (sub-directories), that are directly beneath it in the hierarchy. Entries in a directory file are called links. A link associates a file identifier with a filename. By convention, a directory contains at least two links, **.** (dot) and **..** (dot-dot). The link called dot refers to the directory itself while dot-dot refers to its parent directory. The root directory, which is the top-most node of the hierarchy, has itself as its parent directory. The pathname of the root directory is **/** and the parent directory of the root directory is **/**.

Execution-time Symbolic Constants

The following constants may be used by applications at execution time to determine which optional facilities are present and what actions shall be taken by the implementation in implementation defined circumstances [see **fpathconf**(BA_OS)].

_POSIX_CHOWN_RESTRICTED	If true, and the calling process is not super-user, the **chown** function cannot be used to modify the user ID of a file, and may only be used to modify the group of a file to the effective group ID or one of the supplementary group IDs of the calling process.
_POSIX_NO_TRUNC	If true, pathname components longer than **{NAME_MAX}** generate an error.
_POSIX_VDISABLE	If true, terminal special characters can be disabled.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions. The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process, or one of its ancestors, evolved from a file that had the set user ID bit or set group ID bit set [see **exec**(BA_OS)]. In addition, they can be reset with the **setuid** and **setgid** routines, respectively [see **setuid**(BA_OS)].

Environmental Variables

When a process begins, an array of strings called the *environment* is made available by an exec routine [see **system**(BA_OS)]. By convention, these strings have the form *variable*=*value*, for example, **PATH=:/usr/sbin**. These environmental variables provide a way to make information about an end-user's environment available to programs [see **envvar**(BA_ENV)].

ETSDU

The Expedited Transport Service Data Unit (ETSDU), is the expedited data transmitted over a transport connection and whose identity is preserved from one end of a transport connection to the other (*i.e.*, an expedited message).

File

A file is an object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. File types include regular, character special, block special, FIFO special and directory.

File Access Permissions

Read, write, and execute/search permissions [see **chmod**(BA_OS)] on a file are granted to a process if one or more of the following are true:

- The effective user ID of the process is a user with appropriate permissions (such as a super-user).
- The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the *owner* portion of the file mode is set.
- The effective user ID of the process does not match the user ID of the owner of the file and the effective group ID of the process matches the group of the file and the appropriate access bit of the *group* portion of the file mode is set.
- The effective user ID of the process does not match the user ID of the owner of the file and the effective group ID of the process does not match the group ID of the file and the appropriate access bit of the *other* portion of the file mode is set.

Otherwise, the corresponding permissions are denied.

File Descriptor

A file descriptor is a non-negative integer used to identify a file for the purposes of doing I/O. An open file descriptor is obtained (for example) from a call to the **creat**, **dup**, **fcntl**, **open**, or **pipe** routines.

A file descriptor has associated with it information used in performing I/O on the file: a file pointer that marks the current position within the file where I/O will begin; file status and access modes (*e.g.*, read, write, read/write) [see **open**(BA_OS)]; and close-on-exec flag [see **fcntl**(BA_OS)]. Multiple file descriptors may identify the same file. The file descriptor is used as an argument by such

routines as the **read**, **write**, **ioctl**, and **close** routines.

Filename

Strings consisting of **1** to **{NAME_MAX}** characters may be used to name, for example, a regular file, a special file or a directory. **{NAME_MAX}** must be at least **14**. These characters may be selected from the set of all character values excluding the characters "null" and slash (**/**).

Note that it is generally unwise to use *****, **$**, **?**, **!**, **[**, or **]** as part of a filename because of the special meaning attached to these characters for filename expansion by the command interpreter [see **system**(BA_OS)]. Other characters to avoid are the hyphen, blank, tab, **<**, **>**, backslash, single and double quotes, grave accent, vertical bar, circumflex, curly braces, and parentheses. It is also advisable to avoid the use of non-printing characters in filenames. A filename is sometimes referred to as a pathname component. The interpretation of a pathname component is dependent on the values of **{NAME_MAX}** and **{_POSIX_NO_TRUNC}** associated with the path prefix of that component. If any pathname component is longer than **{NAME_MAX}** and **{_POSIX_NO_TRUNC}** is in effect for the path prefix of that component [see **fpathconf**(BA_OS)], an error condition exists in that implementation. Otherwise, the implementation uses the first **{NAME_MAX}** bytes of the pathname component.

File Times Update

Each file has three associated time values that are updated when file data has been accessed, file data has been modified, or file status has been changed, respectively. These values are returned in the file characteristics structure [see **stat**(BA_OS)].

Many functions in this interface definition that read or write file data or change the file status specify that the appropriate time-related fields are marked for update. At an update point in time, any marked fields are set to the current time and the update marks cleared. Two such update points are when the file is no longer open by any process and when **stat** or **fstat** are performed on the file. Additional update points are unspecified. Updates are not done for files on read-only file systems.

Foreground Process Group

Each session that has established a connection with a controlling terminal distinguishes one process group of the session as the foreground process group of that controlling terminal. The foreground process group has certain privileges when accessing its controlling terminal that are denied to background process groups [see **termio**(BA_DEV)].

Foreground Process Group ID

The foreground process group ID is the process group ID of the foreground process group.

Group ID

Each system user is a member of at least one group. A group is identified by a group ID, which is a non-negative integer that can be contained in an object of type **gid_t**. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, a saved set-group-ID, or one of the supplementary group IDs. When the identity of a group is associated with a file, it is used to verify its access by processes. The group ID of a newly created file is initialized to the effective group ID of the process that created it unless the set-group-ID flag of the file's parent directory is set; in that case, it is initialized to the group ID of the parent directory.

Implementation-specific Symbolic Names

In detailed definitions of components, it is sometimes necessary to refer to symbolic names that are implementation-specific, but which are not necessarily expected to be accessible to an application program. Many of these symbolic names describe boundary conditions and system limits.

In the SVID, for readability, these implementation-specific values are given symbolic names. These names always appear enclosed in curly brackets to distinguish them from symbolic names of other implementation-specific constants that are accessible to application programs by header files. These names are not necessarily accessible to an application-program through a header file, although they may be defined in the documentation for a particular system.

In general, a portable application program should not refer to these symbolic names in its code. For example, an application-program would not be expected to test the length of an argument list given to an exec routine to determine if it was greater than **{ARG_MAX}**. The following is a list of the implementation-specific symbolic names that may be used in System V component definitions:

Name	*Description*
{ARG_MAX}	max. length of argument list to exec
{CHAR_BIT}	number of bits in a **char**
{CHAR_MAX}	max. integer value of a **char**
{SCHAR_MAX}	max. integer value of a **signed char**
{UCHAR_MAX}	max. integer value of a **unsigned char**
{CHILD_MAX}	max. number of processes per user ID
{CLK_TCK}	number of clock ticks per second
{FCHR_MAX}	max. size of a file in bytes
{INT_MAX}	max. decimal value of an **int**
{UINT_MAX}	max. decimal value of an **unsigned int**
{LINK_MAX}	max. number of links to a single file
{LOCK_MAX}	max. number of entries in system lock table
{LONG_BIT}	number of bits in a **long**
{LONG_MAX}	max. decimal value of a **long**
{ULONG_MAX}	max. decimal value of an **unsigned long**
{MAXDOUBLE}	max. decimal value of a **double**
{MAX_CANON}	max. number of bytes in a terminal canonical input line
{MAX_INPUT}	max. number of bytes required as input
{MAX_CHAR}	max. size of character input buffer
{MAXUID}	max. value for a user ID
{MB_LEN_MAX}	max. number of bytes in a multibyte character for any supported locale
{NAME_MAX}	max. number of characters in a filename

{NGROUPS_MAX}	max. number of supplementary group IDs per process
{FILENAME_MAX}	size needed for an array of **char** large enough to hold the longest filename string that can be opened
{OPEN_MAX}	max. number of files a process can have open
{FOPEN_MAX}	max. number of files that can be open simultaneously
{PASS_MAX}	max. number of significant characters in a password
{PATH_MAX}	max. number of characters in a pathname
{PID_MAX}	max. value for a process ID
{PIPE_BUF}	max. number bytes atomic in write to a pipe
{PROC_MAX}	max. number of simultaneous processes, system wide
{SHRT_MAX}	max. decimal value of a **short**
{USHRT_MAX}	max. decimal value of an **unsigned short**
{STD_BLK}	number of bytes in a physical I/O block
{SYS_NMLN}	number of characters in string returned by **uname**
{SYS_OPEN}	max. number of files open on system
{TMP_MAX}	max. number of unique names generated by **tmpnam**
{WORD_BIT}	number of bits in a **word** or **int**
{CHAR_MIN}	min. integer value of a **char**
{SCHAR_MIN}	min. integer value of a **signed char**
{INT_MIN}	min. decimal value of an **int**
{LONG_MIN}	min. decimal value of a **long**
{SHRT_MIN}	min. decimal value of a **short**

Named Stream

A STREAMS-based file descriptor can be attached to any name in the file system namespace by means of the **fattach** routine. This new object is a named stream. All subsequent **opens** and operations on the named stream act on the stream that was associated with the file descriptor until the name is disassociated from the stream by using the **fdetach** routine.

netbuf Structure

The **netbuf** structure is used by many of the library functions and is defined by the **tiuser.h** header file. This structure includes the following members:

```
unsigned int maxlen;   /* max buffer length */
unsigned int len;      /* length of data in buffer */
char *buf;             /* pointer to data buffer */
void *buf;
```

Orphaned Process

An orphaned process is a process whose creator's lifetime has ended.

Orphaned Process Group

An orphaned process group is a process group in which the parent of every member is either itself a member of the group or is not a member of the group's session.

Parent Process ID

The parent process ID of a process is the process ID of its creator, for the lifetime of its creator [see **exit**(BA_OS)]. A new process is created by a currently active process [see **fork**(BA_OS)]. After the creator's lifetime has ended, the parent process ID is set to the process ID of a special system process.

Passive Transport User

A passive transport user is a transport user that listens for an incoming connect indication.

Pathname and Path Prefix

In a **C** program, a pathname is a null-terminated character string starting with an optional slash (**/**), followed by zero or more directory-names separated by slashes, optionally followed by a filename. A null string is undefined and may be considered an error.

A pathname is used to identify a file. It consists of at most, **{PATH_MAX}** bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more filenames separated by slashes. If the pathname refers to a directory, it may also have one or more trailing slashes. Multiple consecutive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes are treated as a single slash.

If a pathname begins with a slash, the path search begins at the root directory. Otherwise, the search begins from the current working directory. If a pathname refers to a directory, it may also have one or more trailing slashes. Multiple consecutive slashes are considered the same as a single slash.

A slash by itself names the root directory. An attempt to create or delete the pathname slash by itself is undefined and may be considered an error.

The meanings of **.** (dot) and **..** (dot-dot) are defined under **directory**.

Persistent Link

A persistent link is a "link" created between a multiplexer and a driver by the **I_PLINK ioctl** request. This differs from a normal link created by the **I_LINK ioctl** request in that a persistent link remains intact even after the file descriptor associated with the stream above the multiplexer has been closed.

Process

A process is an address space and single thread of control that executes within that address space and its required system resources. A process is created by another process issuing the **fork** function. The process that issues the **fork** is known as the parent process, and the new process created by the **fork** is known as the child process.

Process Group

Each process in the system is a member of a process group that is identified by a process group ID. This grouping permits the signaling of related processes. A newly-created process joins the process group of its creator. A process may change its process group via the **setpgid** function [see **setpgid**(BA_OS)].

Process Group ID

Each process group in the system is uniquely identified by a positive integer that can be contained in an object of type **pid_t**, called a process group ID.

Process Group Leader

A process group leader is a process that creates a new process group. The process group ID of a process group is equal to the process ID of the process group leader.

Process Group Lifetime

After a process group is created with the **setpgid** or **setsid** functions, it is considered active. During its lifetime, other processes may join and leave the process group [see **setpgid**(BA_OS)]. The lifetime of the process group ends when the last remaining process in the group either leaves the process group or terminates.

Process ID

Each process in the system is uniquely identified by a positive integer that can be contained in an object of type **pid_t**, called a process ID. A process ID may not be reused by the system until the lifetimes of any process, process group, or session whose IDs are equal to the process ID are ended.

Process Lifetime

After a process is created with a **fork** function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive or zombie state, where certain resources may be returned to the system, although some resources such as process IDs, are still in use. When another process executes a **wait** function for an inactive process, the remaining resources are returned to the system, and the lifetime of the process ends.

Protocol Address

An address, also known as the Transport Service Access Point (TSAP) address, that identifies the transport user.

pseudo-tty

A pseudo-tty consists of a slave side and a master side. The slave side presents a terminal interface to the user and the master side implements the terminal functions as if an actual terminal device were present. Any data written to the slave side is given to the master side as input and vice versa.

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer called a real user ID. Each user is also a member of a group. The group is identified by a positive-integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process. They can be reset with the **`setuid`** and **`setgid`** routines, respectively.

Root Directory and Current Working Directory

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path searches. The root directory of a process need not be the root directory of the root file system [see **`chroot`**(KE_OS)].

Saved Set-user-ID and Saved Set-group-ID

The saved set-user-ID and saved set-group-ID are the values of the effective user ID and effective group ID prior to an exec of a file whose set-user or set-group file mode bit has been set [see **`exec`**(BA_OS)].

Scheduling class

A scheduling class is a process attribute that determines the scheduling policy applied to the process. Every active process in a system has a class associated with it, *i.e.* belongs to a scheduling class.

Session

Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership[see **setsid**(BA_OS)].

Session ID

Each session in the system is uniquely identified by a positive integer that can be contained in an object of type **pid_t**, called a session ID.

Session Leader

A session leader is a process that creates a new session. The session ID of a session is equal to the process ID of the session leader. Session leaders may allocate controlling terminals to their session, thereby becoming controlling processes [see **setsid**(BA_OS)].

Session Lifetime

After a session is created by a session leader, it is considered active. The lifetime of the session ends when the last remaining process in the session either leaves the session or terminates.

Signal

A signal is a mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term signal is also used to refer to the event itself [see **signal**(BA_ENV)].

Special Processes

All special processes are system processes (*e.g.*, a system's swapper process). Certain process IDs are reserved for special processes.

stdio Routines

A set of routines described as Standard I/O (stdio) routines constitute an efficient, user-level I/O buffering scheme. The complete set of stdio routines is shown below [see the definition of **stdio-stream**]. Detailed component definitions of each can be found in either the *Base OS Service Routines* chapter or in the *Base System Library Routines* chapter.

(BA_OS)
clearerr, **fclose**, **fdopen**, **feof**, **ferror**, **fileno**, **fflush**, **fopen**, **fread**, **freopen**, **fseek**, **ftell**, **fwrite**, **popen**, **pclose**, **rewind**.

(BA_LIB)
ctermid, **fgetc**, **fgets**, **fprintf**, **fputc**, **fputs**, **fscanf**, **getchar**, **gets**, **getw**, **printf**, **putc**, **putchar**, **puts**, **putw**, **scanf**, **setbuf**, **setvbuf**, **tempnam**, **tmpnam**, **ungetc**, **vprintf**, **vfprintf**, **vsprintf**.

The Standard I/O routines and constants are declared in the **stdio.h** header file and need no further declaration.

The **stdio.h** header file also defines three symbolic constants used by the stdio routines:

The defined constant **NULL** designates a nonexistent null pointer.

The integer constant **EOF** is returned upon end-of-file or error by most integer functions that deal with streams (see the individual component definitions for details).

The integer constant **BUFSIZ** specifies the size of the buffer required by the **setbuf** routine [see **setbuf**(BA_LIB)].

Any application program that uses the stdio routines must include the **stdio.h** header file.

stdio-stream

A file with associated stdio buffering is called a stdio-stream. A stdio-stream is a pointer to a type **FILE** defined by the **stdio.h** header file. The **fopen** routine [see **fopen**(BA_OS)] creates certain descriptive data for a stream and returns a pointer that identifies the stream in all further transactions with other stdio routines.

Most stdio routines manipulate either a stream created by the function **fopen** or one of three streams that are associated with three files that are expected to be open in the Base System [see **termio**(BA_ENV)]. These three streams are declared in the **stdio.h** header file:

stdin the standard input.

stdout the standard output.

stderr the standard error.

Output streams, with the exception of the standard error stream **stderr**, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream **stderr** is by default unbuffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal immediately; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a newline character is written or terminal input is requested). The **setbuf** and **setvbuf** routines [see **setbuf**(BA_LIB)] may be used to change the stream's buffering strategy.

Stream

A stream is a full duplex connection between a user process and an open device or pseudo-device. The stream itself exists entirely within the kernel and provides a general character I/O interface for user processes. It optionally includes one or more intermediate processing modules that are interposed between the user process end of the stream and the device driver (or pseudo-device driver) end of the stream.

Stream head and stream end

The stream head is the beginning of the stream and is at the kernel/user boundary. This is also known as the upstream end of the stream.

The stream end is the driver end of the stream and is also known as the downstream end of the stream.

Data generated as a result of a system call and destined for the driver end of the stream moves downstream; and data moving from the driver end of the stream toward the stream head is moving upstream. Also, an intermediate Module A is said to be upstream from Module B when it is interposed between Module B and the stream head (upstream) end of the stream, and downstream from Module B when it is between Module B and the driver end of the stream.

STREAMS messages

STREAMS I/O is based on messages. A message may contain a data part, control part, or both. The data part is information that is sent out to a device and the control information is used by the local STREAMS modules. Some messages are used between modules and are not accessible to users. Message types are classified according to their queuing priority and may be normal (non-priority), priority, or high priority messages. A message belongs to a particular priority band that determines its ordering when placed in a queue. Normal messages are always placed at the end of the queue following all other messages in the queue. High priority messages are always placed at the head of a queue but after any other high priority messages already in the queue. Priority messages are always placed after any messages of the same priority or other priority messages but before normal messages. High priority and priority messages are used to send control and data information outside the normal flow of control.

STREAMS module and STREAMS driver

A STREAMS component may be a module or a driver that conforms to the rules specified for STREAMS. A STREAMS device driver or pseudo-device driver is always "opened" and may be "linked" if it is a multiplexing driver. A STREAMS module is any other type of software module such as a line discipline or protocol module and is always "pushed" onto the stream.

STREAMS queue

Each STREAMS module contains two queues, one for messages moving in each direction. A queue structure is defined for STREAMS and is important to the module implementer.

Super-user

The functional implementation means of associating all appropriate privileges to a process. A process is recognized as a super-user process if its effective user ID is 0.

Supplementary Group ID

A process has up to **{NGROUPS_MAX}** supplementary group IDs used in determining file access permissions in addition to the effective group ID. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created, and can be initialized with the **setgroups** function [see **setgroups** in **getgroups**(BA_OS)].

symbolic link

A symbolic link is a special type of file that symbolically represents another file. The contents of a symbolic link are the pathname of the file to which it refers where the referenced file may be any type of file. The use of this mechanism allows directories as well as files to be linked together and permits linking across file systems.

Synchronous Execution

Synchronous execution is the mode of execution in which transport service functions wait for specific events to occur before returning control to the user.

Transport Endpoint

A transport endpoint is the communication path, which is identified by a local file descriptor, between a transport user and a specific transport provider. A transport endpoint is manifested as an open device special file.

Transport Provider

A transport provider is an implementation of a transport protocol that provides the services of the transport layer as defined by the Open Systems Interconnection (OSI) Reference Model. All requests to the transport provider must pass through a transport endpoint.

Transport User

A transport user is a user-level application or protocol that is accessing the services of the transport interface.

TSDU

The Transport Service Data Unit (TSDU), which is the user data transmitted over a transport connection and whose identity is preserved from one end of a transport connection to the other (*i.e.*, a message).

User ID

Each system user is identified by a user ID, which is a non-negative integer that can be contained in an object of type **`uid_t`**. When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or a saved set-user-ID. The user ID of a newly created file is initialized to the effective user ID of the process that created it.

Zombie Process

A zombie process is an inactive process which will be deleted at some later time when its parent process waits for it [see **wait**(BA_OS) and **waitpid**(BA_OS)].

Base Syetem Environment Routines

The following section contains the manual pages for the BA_ENV routines.

NAME

assert: assert.h – verify program assertion

SYNOPSIS

```
#include <assert.h>
```

DESCRIPTION

The <assert.h> header defines the macro assert() and refers to the macro NDEBUG which is not defined in the header. If NDEBUG is defined as a macro name before the inclusion of this header, the assert() macro is defined simply as:

```
#define assert(ignore) ((void) 0)
```

otherwise, the macro behaves as described in assert(BA_LIB).

The assert() macro is implemented as a macro, not as a function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

SEE ALSO

assert(BA_LIB).

LEVEL

Level 1.

NAME

cpio: cpio.h – cpio archive values

SYNOPSIS

```
#include <cpio.h>
```

DESCRIPTION

Values needed by the `c_mode` field in the header of the cpio archive format are described by:

Name	Description	Value (octal)
C_IRUSR	read by owner	0000400
C_IWUSR	write by owner	0000200
C_IXUSR	execute by owner	0000100
C_IRGRP	read by group	0000040
C_IWGRP	write by group	0000020
C_IXGRP	execute by group	0000010
C_IROTH	read by others	0000004
C_IWOTH	write by others	0000002
C_IXOTH	execute by others	0000001
C_ISUID	set uid	0004000
C_ISGID	set gid	0002000
C_ISVTX	reserved	0001000
C_ISDIR	directory	0040000
C_ISFIFO	FIFO	0010000
C_ISREG	regular file	0100000
C_ISBLK	block special	0060000
C_ISCHR	character special	0020000
C_ISCTG	reserved	0110000
C_ISLNK	reserved	0120000
C_ISSOCK	reserved	0140000

The header defines the symbolic constant:

```
MAGIC     "070707"
```

SEE ALSO

cpio(BU_CMD).

LEVEL

Level 1.

NAME

ctype: ctype.h – character types

SYNOPSIS

```
#include <ctype.h>
```

DESCRIPTION

The `<ctype.h>` header declares the following as functions or macros:

```
isalnum()   isgraph()   isupper()
isalpha()   islower()   isxdigit()
isascii()   isprint()   toascii()
iscntrl()   ispunct()   tolower()
isdigit()   isspace()   toupper()
```

The following are declared as macros:

```
_toupper()
_tolower()
```

LEVEL

Level 1.

NAME

dirent: dirent.h – format of directory entries

SYNOPSIS

```
#include <dirent.h>
```

DESCRIPTION

The <dirent.h> header defines the following data type through typedef:

DIR A type representing a directory stream.

Defines the structure dirent which includes the following members:

```
ino_t   d_ino;              /* file serial number */
char    d_name[{NAME_MAX}];  /* name of entry */
```

The type ino_t is defined in <sys/types.h> [see types(BA_ENV)].

The character array d_name is of unspecified size, but the number of characters preceding the terminating null character shall not exceed {NAME_MAX}.

The following are declared as the functions:

```
closedir()   rewinddir()
opendir()    seekdir()
readdir()    telldir()
```

SEE ALSO

directory(BA_OS), types(BA_ENV).

LEVEL

Level 1.

NAME

envvar – environment variables

DESCRIPTION

When a process begins execution, exec routines make available an array of strings called the environment [see exec(BA_OS)]. By convention, these strings have the form `variable=value`, for example, `PATH=/sbin:/usr/sbin`. These environmental variables provide a way to make information about a program's environment available to programs. The following environmental variables can be used by applications and are expected to be set in the target run-time environment.

Variable *Use*

`HOME` Full pathname of the user's home directory, the user's initial working directory [see passwd(BA_ENV)].

`PATH` Colon-separated, ordered list of pathnames that determines the search sequence used in locating files [see system(BA_OS)].

`LANG` The string used to specify localization information that allows users to work with different national conventions. The `setlocale()` function [see setlocale(BA_OS)] looks for the `LANG` environment variable when it is called with `""` as the *locale* argument. `LANG` is used as the default locale if the corresponding environment variable for a particular category is unset.

For example, when `setlocale()` is invoked as

```
setlocale(LC_CTYPE, "")
```

`setlocale`() will query the `LC_CTYPE` environment variable first to see if it is set and non-null. If `LC_CTYPE` is not set or null, then `setlocale()` will check the `LANG` environment variable to see if it is set and non-null. If both `LANG` and `LC_CTYPE` are unset or null, the default `C` locale will be used to set the `LC_CTYPE` category.

Most commands will invoke

```
setlocale(LC_ALL, ""),
```

prior to any other processing. This allows the command to be used with different national conventions by setting the appropriate environment variables.

The following environment variables are supported to correspond with each category of `setlocale()`:

`LC_COLLATE` This category specifies the collation sequence being used. This category affects `strcoll()` and `strxfrm()` [see strcoll(BA_LIB) and strxfrm(BA_LIB), respectively].

`LC_CTYPE` This category specifies character classification, character conversion, and widths of multibyte characters. The default `C` locale corresponds to the 7-bit ASCII character set. This category affects `ctype()` and `mbchar()`

[see ctype(BA_LIB) and mbchar(BA_LIB), respectively].

`LC_MESSAGES` This category specifies the language of the message database being used. For example, an application may have one message database with French messages, and another database with German messages [see gettxt(BA_LIB)].

`LC_MONETARY` This category specifies the monetary symbols and delimiters used for a particular locale. This category affects `localeconv()` [see localeconv(BA_LIB)].

`LC_NUMERIC` This category specifies the decimal and thousandths delimiters. The default `C` locale corresponds to `.` as the decimal delimiter and no thousands delimiter. This category affects `localeconv()`, `printf()` [see printf(BA_LIB)], `scanf()` [see scanf(BA_LIB)] and `strtod()` [see strtod(BA_LIB)].

`LC_TIME` This category specifies date and time formats. The default `C` locale corresponds to U.S. date and time formats. This category affects `strftime()` [see strftime(BA_LIB)].

`SEV_LEVEL` Define severity levels and associates and print strings with them in standard format error messages [see fmtmsg(BA_LIB)].

`MSGVERB` Controls which standard format message components `fmtmsg()` selects when messages are displayed to stderr[see also fmtmsg(BA_LIB)].

`NETPATH` A colon-separated list of network identifiers. A network identifier is a character string used by the Network Selection component of the system to provide application-specific default network search paths. A network identifier must consist of non-NULL characters and must have a length of at least 1. No maximum length is specified. Network identifiers are normally chosen by the system administrator.

`NLSPATH` Contains a sequence of templates which `catopen()` uses when attempting to locate message catalogues. Each template consists of an optional prefix, one or more substitution fields, a filename and an optional suffix.

For example:

```
NLSPATH="/system/nlslib/%N.cat"
```

defines that `catopen()` should look for all message catalogues in the directory `/system/nlslib`, where the catalogue name should be constructed from the *name* parameter passed to `catopen()`, `%N`, with the suffix `.cat`.

Substitution fields consist of a % symbol, followed by a single-letter keyword. The following keywords are currently defined:

%N	The value of the name parameter passed to catopen().
%L	The value of LANG.
%l	The language element from LANG.
%t	The territory element from LANG.
%c	The codeset element from LANG.
%%	A single % character.

An empty string is substituted if the specified value is not currently defined. The separators "_" and "." are not included in %t and %c substitutions.

Templates defined in NLSPATH are separated by colons (:). A leading colon or two adjacent colons (::) is equivalent to specifying %N.

For example:

```
NLSPATH=":%N.cat:/nlslib/%L/%N.cat"
```

indicates to catopen() that it should look for the requested message catalogue in *name*, *name*.cat and /nlslib/$LANG/*name*.cat.

PATH The sequence of directory prefixes that are applied in searching for a file known by an incomplete path name. The prefixes are separated by colons (:).

TERM The kind of terminal for which output is to be prepared. This information is used by commands which may exploit special capabilities of that terminal.

TZ Time zone information.
The contents of the environment variable named TZ are used by the functions ctime(), localtime(), strftime() and mktime() to override the default timezone. If the first character of TZ is a colon (:), the behavior is implementation defined, otherwise TZ has the form:

std offset [dst [offset], [start [/ time], end [/ time]]]

Where:

std and *dst*
Three or more bytes that are the designation for the standard (*std*) and summer (*dst*) timezones. Only *std* is required, if *dst* is missing, then summer time does not apply in this locale. Upper- and lower-case letters are explicitly allowed. Any characters except a leading colon (:), digits, a comma (,), a minus (-) or a plus (+) are allowed.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The offset has the form:

hh [: mm [: ss]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst* , summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between 0 and 24, and the minutes (and seconds) if present between 0 and 59. Out of range values may cause unpredictable behaviour. If preceded by a ''–'', the timezone is east of the Prime Meridean; otherwise it is west (which may be indicated by an optional preceding ''+'' sign).

start / *time*, *end* / *time*
Indicates when to change to and back from summer time, where *start/time* describes when the change from standard time to summer time occurs, and *end/time* describes when the change back happens. Each *time* field describes when, in current local time, the change is made.

The formats of *start* and *end* are one of the following:

J*n* The Julian day *n* ($1 \le n \le 365$). Leap days are not counted. That is, in all years, February 28 is day 59 and March 1 is day 60. It is impossible to refer to the occasional February 29.

n The zero-based Julian day ($0 \le n \le 365$). Leap days are counted, and it is possible to refer to February 29.

M*m.n.d*
The d^{th} day, ($0 \le d \le 6$) of week *n* of month *m* of the year ($1 \le n \le 5$, $1 \le m \le 12$), where week 5 means ''the last *d*-day in month *m*'' which may occur in either the fourth or the fifth week). Week 1 is the week in which the first day of the month falls. Day zero is Sunday.

Implementation specific defaults are used for *start* and *end* if these optional fields are not given.

The *time* has the same format as *offset* except that no leading sign (''–'' or ''+'') is allowed. The default, if *time* is not given is 02:00:00.

SEE ALSO

ctype(BA_LIB), exec(BA_OS), filsys(BA_ENV), getenv(BA_LIB), gettxt(BA_LIB), localeconv(BA_LIB), mbchar(BA_LIB), printf(BA_LIB), putenv(BA_LIB), setlocale(BA_OS), strcoll(BA_LIB), strftime(BA_LIB), strtod(BA_LIB), strxfrm(BA_LIB), system(BA_OS).

FUTURE DIRECTIONS

The number in TZ will be defined as an optional minus sign followed by two hour digits and two minute digits, *hhmm*, in order to represent fractional time-zones.

LEVEL

Level 1.

NAME

errors – error code and condition definitions

SYNOPSIS

```
#include <errno.h>

errno
```

DESCRIPTION

The numerical value represented by the symbolic name of an error condition is assigned to `errno` for errors that occur when executing a system service routine or general library routine.

To be consistent with the C Standard, the interface definition of `errno` has been change in the SIVD, Fourth Edition. Programs should obtain the value of `errno` by including `<errno.h>`.

The macro `errno` expands to a modifiable *lvalue* that has type `int`, the value of which is set to a positive error number by several library functions. `errno` need not be the identifier of an object, *e.g.*, it might expand to a modifiable *lvalue* resulting from a function call. It is unspecified whether `errno` is a macro or an identifier declared with external linkage. If an `errno` macro definition is suppressed to access an actual object, or if a program defines an identifier with the name `errno`, the behavior is undefined.

The component definitions given in the *BASE OS SERVICE ROUTINES* chapter and in the *BASE LIBRARY ROUTINES* chapter, list possible error conditions for each routine and the meaning of the error in that context. The order in which possible errors are listed is not significant and does not imply precedence. The value of `errno` should be checked only after an error has been indicated; that is, when the return value of the component indicates an error, and the component definition specifies that `errno` be set. The `errno` value `0` is reserved; no error condition is equal to zero. An application that checks the value of `errno` must include the `<errno.h>` header file.

Additional error conditions may be defined by Extensions to the Base System or by particular implementations.

The following list describes the general meaning of each error:

`E2BIG` Argument list is too long.
An argument list longer than `{ARG_MAX}` bytes was presented to a member of the exec family of routines.

`EACCES` Permission is denied.
An attempt was made to access a resource in a way forbidden by the protection system.

`EAGAIN` Resource is temporarily unavailable; try again later.
For example, the `fork( )` routine failed because the process table of the system is full.

`EBADF` File number is bad.
Either a file descriptor refers to no open file, or a read (respectively, write) request was made to a file that is open only for writing (respectively, reading).

EBADMSG Bad message.
During a `read( )`, `getmsg( )`, or `ioctl( ) I_RECVFD` system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:
`read( )` - control information or a passed file descriptor.
`getmsg( )` - passed file descriptor.
`ioctl( )` - control or data information.

EBUSY Device or resource busy or unavailable.
An attempt was made to make use of a system resource that is not currently available, as it is being used by another process in a manner that would have conflicted with the request being made by this process. For example attempting to mount a device that was already mounted or to unmount a device on which there is an active file (open file, current directory, mounted on file, active text segment).

ECANCELED Asynchronous I/O canceled.
The requested I/O was canceled before the I/O completed because of aio_cancel.

ECHILD No child processes.
An attempt was made to obtain the status of a child process or processes, by a process that had no existing child process in the appropriate state.

EDEADLK Deadlock avoided.
The request would have caused a deadlock; the situation was detected and avoided.

EDOM Math argument.
The argument of a function in the math package is out of the domain of the function.

EEXIST File exists.
An existing file was mentioned in an inappropriate context (*e.g.*, a call to the `link( )` routine).

EFAULT Address is bad.
The system encountered a hardware fault in attempting to use an argument of a routine. For example, `errno` potentially may be set to `EFAULT` any time an invalid address is passed a routine that takes a pointer argument if the system can detect the condition. Because systems differ in their ability to reliably detect a bad address, on some implementations passing a bad address to a routine will result in undefined behavior.

EFBIG File is too large.
The size of a file exceeded the maximum file size limit [see getrlimit(BA_OS)].

EIDRM Identifier removed.
An identifier has been removed from the system.

EINPROGRESS
The operation requested is now in progress.
An operation that takes a long time to complete was attempted on a non-blocking object.

EINTR Interrupted system service.
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system service routine. If execution is resumed after processing the signal, it will appear as if the interrupted routine returned this error condition.

EINVAL Invalid argument.
An invalid argument (*e.g.*, unmounting a non-mounted device; mentioning an undefined signal in a call to the `signal( )` or `kill( )` routine). Also set by math routines.

EIO I/O error.
Some physical I/O error has occurred, or access to controlling terminal denied to a background process. For physical I/O errors, this error may, in some cases, occur on a call following the one to which it actually applies.

EISDIR Is a directory.
An invalid operation on a directory was attempted. For example, an attempt was made to write on a directory.

ELIBACC Reserved for system use.

ELIBBAD Reserved for system use.

ELIBEXEC Reserved for system use.

ELIBMAX Reserved for system use.

ELIBSCN Reserved for system use.

ELOOP Too many levels of symbolic links.
Too many symbolic links were encountered in translating pathname.

EMFILE Too many open files in a process.
No process may have more than {OPEN_MAX} file descriptors open at a time.

EMLINK Too many links.
An attempt was made to make more than the maximum number of links {LINK_MAX} to a file.

ENAMETOOLONG
if the filename is too long. if the length of a pathname exceeds {PATH_MAX}, or the length of a path component exceeds {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

ENFILE Too many open files in the system.
The system file table is full (*i.e.*, {SYS_OPEN} files are open, and temporarily no more *opens* can be accepted).

ENODEV No such device.
An inappropriate operation to a device is attempted. (*e.g.*, read a write-only device).

ENOENT No such file or directory.
A filename is specified and the file should exist but doesn't, or one of the directories in a pathname does not exist.

ENOEXEC Exec format error.
A request is made to execute a file which, despite appropriate permissions, does not start with a valid format.

ENOLCK No locks available.
A system-imposed limit on the number of simultaneous file and record locks was reached and no more are available at that time.

ENOLOAD Failure in loading a loadable exec module.
An attempt was made to dynamically load an executable module and the attempt failed.

ENOMEM Not enough space.
During execution of an exec routine, a program asks for more space than the system is able to supply. This is not a temporary condition until other processes release resources. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during execution of the `fork( )` routine.

ENOMSG No message of the desired type.
The message queue does not contain a message of the required type.

ENOPKG Package not installed.
An attempt was made to use a system call from a package which has not been installed.

ENOSPC No space is left on the device.
While writing a regular file or creating a directory entry, no free space is left on the device.

ENOSR No stream resources.
Insufficient STREAMS memory resources are available to perform a STREAMS related system call. This is a temporary condition; one may recover from it if other processes release resources.

ENOSTR Not a stream.
`putmsg( )` or `getmsg( )` system call is attempted on a file descriptor that is not a STREAMS device.

ENOSYS Operation not applicable.
A non-existing system operation is requested from a file system type, or an attempt was made to use a function that is not available in this implementation.

`ENOTBLK` Block device is required.
A non-block file is mentioned where a block device is required (*e.g.*, in a call to the `mount( )` routine).

`ENOTDIR` Not a directory.
A non-directory is specified where a directory is required (*e.g.* in a path-prefix or as an argument to the `chdir( )` routine).

`ENOTEMPTY` Directory not empty.
A directory with entries other than `.` and `..` was supplied when an empty directory was expected.

`ENOTTY` Not a character device.
A call is made to a character special device system server routine, specifying a file that is not a character special device.

`ENXIO` No such device or address.
I/O on a special file refers to a subdevice which does not exist, or exists beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

`EOVERFLOW` Reserved for system use.

`EPERM` Operation not permitted.
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or a process with appropriate privileges. It is also returned for attempts by processes to do things allowed only to processes with appropriate privileges.

`EPIPE` Broken pipe.
A write on a pipe for which no process can read the data. This condition generates a `SIGPIPE` signal; the error is returned if the signal is ignored.

`EPROTO` Protocol error.
Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

`ERANGE` Result is too large.
The value of a function in the math package is not representable within machine precision.

`ERESTART` Reserved for system use.

`EROFS` Read-only file system.
An attempt to modify a file or directory is made on a device mounted read-only.

`ESPIPE` Illegal seek.
A call to the `lseek()` routine is issued to a pipe or a named STREAMS pipe [see lseek(BA_OS)].

`ESRCH` No such process.
No process can be found corresponding to the specified search criteria.

ESTRPIPE Reserved for system use.

ETXTBSY Text file busy.
An attempt made to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing a pure-procedure program that is being executed.

ETIME Stream `ioctl()` timeout.
The timer set for a STREAMS `ioctl()` call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or a timeout value that is too short for the specific operation. The status of the `ioctl()` operation is indeterminate.

EXDEV Cross-device link.
A link to a file on another device is attempted.

USAGE

Some routines do not have an error return value. Because no routine sets `errno` to zero, an application may, in this case, set `errno` to zero, call a routine, and then if the component definition specifies that `errno` be set, check whether errno has been set to indicate an error. A routine can save the value of `errno` on entry and then set it to zero, as long as the original value is restored if `errno` is still zero just before return.

SEE ALSO

chdir(BA_OS), exec(BA_OS), fork(BA_OS), getmsg(BA_OS), ioctl(BA_OS), kill(BA_OS), link(BA_OS), lseek(BA_OS), mount(BA_OS), ptrace(KE_OS), putmsg(BA_OS), read(BA_OS), ulimit(BA_OS), wait(BA_OS).

LEVEL

Level 1.

NAME

fcntl: fcntl.h – file control options

SYNOPSIS

```
#include <fcntl.h>
```

DESCRIPTION

The <fcntl.h> header defines the following requests and arguments for use by the functions fcntl() [see fcntl(BA_OS)] and open() [see open(BA_OS)].

Values for *cmd* used by fcntl() (the following values are unique):

F_DUPFD	Duplicate file descriptor
F_GETFD	Get file descriptor flags
F_SETFD	Set file descriptor flags
F_GETFL	Get file status flags
F_SETFL	Set file status flags
F_GETLK	Get record locking information
F_SETLK	Set record locking information
F_SETLKW	Set record locking information; wait if blocked

File descriptor flags used for fcntl():

FD_CLOEXEC	Close the file descriptor upon execution of an exec function [see exec(BA_OS)]

Values for l_type used for record locking with fcntl() (the following values are unique):

F_RDLCK	Shared or read lock
F_UNLCK	Unlock
F_WRLCK	Exclusive or write lock

The following three sets of values are bitwise distinct:
Values for oflag used by open():

O_CREAT	Create file if it does not exist
O_EXCL	Exclusive use flag
O_NOCTTY	Do not assign controlling tty
O_TRUNC	Truncate flag

File status flags used for open() and fcntl():

O_APPEND	Set append mode
O_NONBLOCK	Non-blocking mode
O_SYNC	Synchronous writes

Mask for use with file access modes:

O_ACCMODE	Mask for file access modes

File access modes used for `open()` and `fcntl()`:

`O_RDONLY`	Open for reading only
`O_RDWR`	Open for reading and writing
`O_WRONLY`	Open for writing only

The structure `flock` describes a file lock. It includes the following members:

```
short   l_type;     /* Type of lock */
short   l_whence;   /* Flag for starting offset */
off_t   l_start;    /* Relative offset in bytes */
off_t   l_len;      /* Size; if 0 then until EOF */
pid_t   l_pid;      /* Process ID of the process holding
                       the lock; returned with F_GETLK */
```

The following are declared as either functions or macros:

```
creat()   fcntl()
open()
```

SEE ALSO

creat(BA_OS), exec(BA_OS), fcntl(BA_OS), open(BA_OS).

LEVEL

Level 1.

NAME

file system – directory tree structure

DESCRIPTION

Directory Tree Structure

The file system on any System V operating system is a tree-like structure, and is divided into a "root" file system and a collection of mountable file systems.

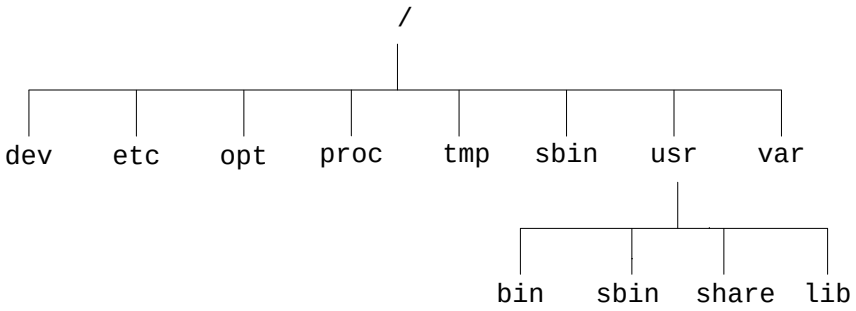
All System V conforming systems must have a "root" (`/`), a "user" (`/usr`) and a "var" (`/var`) subtree accessible to user-level programs. The user, root and var subtrees may or may not be different physical file systems, but their appearance to user programs will always be the same.

The root file system contains machine-specific information (*i.e.*, system data files, log files, *etc.*) and files necessary to boot and run the system.

The directory `/usr` of the root file system is the point of access to the `/usr` subtree, whether it is a real, mounted file system or a subtree of the root file system. All files under the `/usr` directory can be shared between machines of the same architecture, while all files under `/usr/share` can be shared between all machines of the same and disparate architectures.

The directory `/var` is the point of access to the `/var` subtree, whether it is a real, mounted file system or a subtree of the root file system. The `/var` subtree contains files that vary in size and presence during normal system operations, including logging, accounting and temporary files created by the system and applications.

Below is a diagram of the minimal directory tree structure expected to be on any System V operating system.



The following guidelines apply to the contents of these directories:

`/dev`, `/etc`, `/proc`, `/tmp`, `/sbin`, and `/usr/sbin`
: primarily for the use of the system. Most applications should never *create* files in any of these directories, though they may read and execute them. Applications, as well as the system, can use `/usr/bin` and `/var`.

`/dev`
: holds special device files.

`/etc`	holds system data files, such as `/etc/passwd`.
`/opt`	root directory for add-on application packages. For example, `/opt/x` would contain the root of the directory tree for application `x`. Application `x` should place varying files (such as log files and temporary files) in `/var/opt/x`.
`/proc`	place holder for the `proc` file system type.
`/tmp`	holds temporary files created by utilities in `/sbin` and by other system processes.
`/sbin`	holds executable system commands (utilities), if any, needed to bring the system up to a usable state.
`/usr/bin`	holds (user-level) executable application and system commands.
`/usr/lib`	holds libraries and machine architecture-dependent databases.
`/usr/sbin`	holds the bulk of executable system commands (utilities).
`/usr/share`	holds machine-architecture independent database files (such as manual page files). These files many be shared between machines of different hardware types.
`/var`	holds system varying files, such as log files and temporary files.

Applications should install or create files only in designated places within the tree. The primary locations are the `/opt` and `/var/opt` subtrees. Temporary files should always be created using the library routines provided for this purpose [see tmpnam(BA_LIB), `tempnam()` in tmpnam(BA_LIB), tmpfile(BA_LIB), and mktemp(BA_LIB), for example].

Some extensions to the Base System will have additional requirements on the tree structure when the extension is installed on a system. Directory tree requirements specific to an extension will be identified when the extension is defined in detail.

System Data Files

The Base System Definition specifies only these system-resident data files:

```
/etc/group
/etc/passwd
/etc/profile
```

The `/etc/passwd` and `/etc/profile` files are owned by the system and are readable but not writable by ordinary users.

`/etc/passwd` is a generally useful file, readable by applications, that makes available to application programs some basic information about end-users on a system. It has one entry for each user. Minimally, each user's entry contains a string that is the name by which the user is known on the system, a numerical user-ID, and the home directory or initial working directory of the user. [See passwd(BA_ENV) for file format and content details.]

Conventionally, the information in this file is used during the initialization of the environment for a particular user. However, the `/etc/passwd` file is also useful as a database with a standard format containing information about users, which can be used independently of the mechanisms that maintain the data file.

The `/etc/profile` file may contain a string assignment of the `PATH` and `TZ` variables [see envvar(BA_ENV)].

SEE ALSO

envvar(BA_ENV), passwd(BA_ENV).

LEVEL

Level 1.

NAME

float: float.h – numerical limits

SYNOPSIS

```
#include <float.h>
```

DESCRIPTION

The `<float.h>` header provides for the following constants.

The rounding mode for floating point addition is characterized by the value of `FLT_ROUNDS`:

-1	indeterminable
0	toward zero
1	to nearest
2	toward positive infinity
3	toward negative infinity

All other values for `FLT_ROUNDS` characterize implementation-defined behavior.

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

```
#define DBL_DIG                          10
#define DBL_MANT_DIG
#define DBL_MAX_10_EXP                  +37
#define DBL_MAX_EXP
#define DBL_MIN_10_EXP                  -37
#define DBL_MIN_EXP
#define FLT_DIG                           6
#define FLT_MANT_DIG
#define FLT_MAX_10_EXP                  +37
#define FLT_MAX_EXP
#define FLT_MIN_10_EXP                  -37
#define FLT_MIN_EXP
#define FLT_RADIX                         2
#define LDBL_DIG                         10
#define LDBL_MANT_DIG
#define LDBL_MAX_10_EXP                 +37
#define LDBL_MAX_EXP
#define LDBL_MIN_10_EXP                 -37
#define LDBL_MIN_EXP
```

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal to or greater than those shown.

```
#define DBL_MAX                       1E+37
#define FLT_MAX                       1E+37
#define LDBL_MAX                      1E+37
```

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal to or less than those shown.

```
#define DBL_EPSILON                    1E-9
#define DBL_MIN                       1E-37
#define FLT_EPSILON                    1E-5
```

```
#define FLT_MIN                      1E-37
#define LDBL_EPSILON                  1E-9
#define LDBL_MIN                     1E-37
```

The value of `FLT_RADIX` shall be a constant expression suitable for use in preprocessing directives. Values that need not be constant expressions shall be supplied for all other components.

LEVEL

Level 1.

NAME

ftw: ftw.h – file tree traversal

SYNOPSIS

```
#include <ftw.h>
```

DESCRIPTION

The `<ftw.h>` header defines codes for the third argument to the user-supplied function which is passed as the second argument to `ftw()` [see ftw(BA_LIB)]:

`FTW_F`	File
`FTW_D`	Directory
`FTW_DNR`	Directory without read permission
`FTW_NS`	Unknown type, `stat()` failed

Declares the following as a function or a macro:

```
ftw()   nftw()
```

SEE ALSO

ftw(BA_LIB).

LEVEL

Level 1.

NAME

group – group file

DESCRIPTION

The file `group` contains for each group the following information:

group name
encrypted password
numerical group ID
comma-separated list of all users allowed in the group

The file `group` is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory `/etc`. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

During user identification and authentication, the supplementary group access list is initialized sequentially from information in this file. If a user is in more groups than the system is configured for, `{NGROUPS_MAX}`, subsequent group specifications will be ignored.

FILES

`/etc/group`

SEE ALSO

groups(AU_CMD), passwd(AU_CMD), getgroups(BA_OS), initgroups(BA_LIB)

LEVEL

Level 1.

NAME

grp: grp.h – group structure

SYNOPSIS

```
#include <grp.h>
```

DESCRIPTION

The <grp.h> header declares struct group which includes the following members:

```
char   *gr_name;   /* name */
gid_t  gr_gid;     /* numerical group ID */
char   **gr_mem;   /* pointer to a null terminated
                      array of character pointers
                      to member names */
```

The following are declared as either a function or macro:

```
getgrgid()  getgrnam()
```

SEE ALSO

getgrent(BA_LIB).

LEVEL

Level 1.

NAME

langinfo: langinfo.h – language information constants

SYNOPSIS

```
#include <langinfo.h>
```

DESCRIPTION

The `<langinfo.h>` header contains the constants used to identify items of langinfo data [see nl_langinfo(BA_LIB)]. The mode of the constants is given in `<nl_types.h>` [see nl_types(BA_ENV)].

The entries under the Category column of the table below indicate in which `setlocale()` category each item is defined [see setlocale(BA_OS)].

The following constants are defined on all systems:

Constant	Category	Meaning
D_T_FMT	LC_TIME	string for formatting date and time
D_FMT	LC_TIME	date format string
T_FMT	LC_TIME	time format string
AM_STR	LC_TIME	Ante Meridiem affix
PM_STR	LC_TIME	Post Meridiem affix
DAY_1	LC_TIME	name of the first day of the week (e.g., Sunday)
DAY_2	LC_TIME	name of the second day of the week (e.g., Monday)
DAY_3	LC_TIME	name of the third day of the week (e.g., Tuesday)
DAY_4	LC_TIME	name of the fourth day of the week (e.g., Wednesday)
DAY_5	LC_TIME	name of the fifth day of the week (e.g., Thursday)
DAY_6	LC_TIME	name of the sixth day of the week (e.g., Friday)
DAY_7	LC_TIME	name of the seventh day of the week (e.g., Saturday)
ABDAY_1	LC_TIME	abbreviated name of the first day of the week
ABDAY_2	LC_TIME	abbreviated name of the second day of the week
ABDAY_3	LC_TIME	abbreviated name of the third day of the week
ABDAY_4	LC_TIME	abbreviated name of the fourth day of the week
ABDAY_5	LC_TIME	abbreviated name of the fifth day of the week
ABDAY_6	LC_TIME	abbreviated name of the sixth day of the week
ABDAY_7	LC_TIME	abbreviated name of the seventh day of the week
MON_1	LC_TIME	name of the first month in the Gregorian calendar
MON_2	LC_TIME	name of the second month
MON_3	LC_TIME	name of the third month
MON_4	LC_TIME	name of the fourth month
MON_5	LC_TIME	name of the fifth month
MON_6	LC_TIME	name of the sixth month
MON_7	LC_TIME	name of the seventh month
MON_8	LC_TIME	name of the eighth month
MON_9	LC_TIME	name of the ninth month
MON_10	LC_TIME	name of the tenth month
MON_11	LC_TIME	name of the eleventh month
MON_12	LC_TIME	name of the twelfth month
ABMON_1	LC_TIME	abbreviated name of the first month

Constant	Category	Meaning
ABMON_2	LC_TIME	abbreviated name of the second month
ABMON_3	LC_TIME	abbreviated name of the third month
ABMON_4	LC_TIME	abbreviated name of the fourth month
ABMON_5	LC_TIME	abbreviated name of the fifth month
ABMON_6	LC_TIME	abbreviated name of the sixth month
ABMON_7	LC_TIME	abbreviated name of the seventh month
ABMON_8	LC_TIME	abbreviated name of the eighth month
ABMON_9	LC_TIME	abbreviated name of the ninth month
ABMON_10	LC_TIME	abbreviated name of the tenth month
ABMON_11	LC_TIME	abbreviated name of the eleventh month
ABMON_12	LC_TIME	abbreviated name of the twelfth month
RADIXCHAR	LC_NUMERIC	radix character
THOUSEP	LC_NUMERIC	separator for thousands
YESSTR	LC_ALL	affirmative response for yes / no queries
NOSTR	LC_ALL	negative response for yes / no queries
CRNCYSTR	LC_MONETARY	currency symbol, preceded by – if the symbol should appear before the value, + if the symbol should appear after the value, or . if the symbol should replace the radix character

Declares the following as a function:

```
nl_langinfo()
```

SEE ALSO

nl_langinfo(BA_LIB), nl_types(BA_ENV), setlocale(BA_OS).

LEVEL

Level 1.

NAME

limits: limits.h – implementation specific constants

SYNOPSIS

```
#include <limits.h>
```

DESCRIPTION

The `<limits.h>` header defines various names which are used throughout the descriptive text of the System V Interface Definition. Different categories of names are described in the tables below.

The names represent various limits on resources which the system imposes on applications.

Implementations may choose any appropriate value for each limit, provided it is not more restrictive than the values listed in the column headed ''Minimum Acceptable Value'' in the table below.

Applications should not assume any particular value for a limit. To achieve maximum portability, an application should not require more resource than the quantity listed in the ''Minimum Acceptable Value'' column. However, an application wishing to avail itself of the full amount of a resource available on an implementation may make use of the value given in `<limits.h>` on that particular system, by using the symbolic names listed in the first column of the table. It should be noted, however, that many of the listed limits are not invariant, and at run-time, the value of the limit may differ from those given in this header, for the following reasons: the limit is pathname dependent and the limit differs between the compile and run-time machines.

For these reasons, an application may use the `fpathconf()` [see fpathconf(BA_OS)], `pathconf()` [see `pathconf()` in fpathconf(BA_OS)] and `sysconf()` [see sysconf(BA_OS)] functions to determine the actual value of a limit at run-time.

The items in the list ending in ''_MIN'' give the most negative values that the mathematical types are guaranteed to be capable of representing. Numbers of a more negative value may be supported on some systems, as indicated by the `<limits.h>` header on the system, but applications requiring such numbers are not guaranteed to be portable to all systems.

The symbol * in the ''Minimum Acceptable Value'' column indicates that there is no guaranteed value across all compliant systems.

The definition for any of the following names may be omitted from `<limits.h>` if the actual value of the limit is indeterminate but equal to or greater than the stated minimum. Applications should therefore only use these symbols in code conditionally compiled on the existence of the symbol, or in calls to `fpathconf()`, `pathconf()` or `sysconf()`.

Name	Description	Minimum Acceptable Value
ARG_MAX	Max length of argument to the exec functions including environment data	_POSIX_ARG_MAX
CHILD_MAX	Max number of processes per user ID	_POSIX_CHILD_MAX
LINK_MAX	Max number of links to a single file	_POSIX_LINK_MAX
MAX_CANON	Max number of bytes in a terminal canonical input line	_POSIX_MAX_CANON
MAX_INPUT	Max number of bytes allowed in a terminal input queue	_POSIX_MAX_INPUT
MB_LEN_MAX	Max number of bytes in a multibyte character, for any supported locale	1
NAME_MAX	Max number of characters in a filename (not including terminating null)	_POSIX_NAME_MAX
OPEN_MAX	Max number of files that one process can have open at any one time	_POSIX_OPEN_MAX
PASS_MAX	Max number of significant characters in a password (not including terminating null)	8
PATH_MAX	Max number of characters in a pathname (not including terminating null)	_POSIX_PATH_MAX
PIPE_BUF	Max number bytes that is guaranteed to be atomic when writing to a pipe	_POSIX_PIPE_BUF

The following constant will always be defined in `<limits.h>` and will also be available from `sysconf()`.

Name	Description	Minimum Acceptable Value
NGROUPS_MAX	Max number of simultaneous supplementary group IDs per process	_POSIX_NGROUPS_MAX

The following constants will always be defined in <limits.h>

Name	Description	Minimum Acceptable Value
NL_ARGMAX	Max value of "digit" in calls to the printf() and scanf() functions	9
NL_LANGMAX	Max number of bytes in a LANG name	14
NL_MSGMAX	Max message number	32 767
NL_NMAX	Max number of bytes in N-to-1 mapping characters	*
NL_SETMAX	Max set number	255
NL_TEXTMAX	Max number of bytes in a message string	2048
NZERO	default process priority	20
TMP_MAX	Max number of unique names generated by tmpnam()	10 000

The following constants are specified by POSIX 1003.1-1988 and will always be defined in <limits.h>. They are invariant:

Name	Description	Value
_POSIX_ARG_MAX	The length of the argument strings for the exec functions in bytes, including environment data	4 096
_POSIX_CHILD_MAX	The number of simultaneous processes per real user ID.	6
_POSIX_LINK_MAX	the value of a file's link count.	8
_POSIX_MAX_CANON	The number of bytes in a terminal canonical input queue	255
_POSIX_MAX_INPUT	The number of bytes for which space will be available in a terminal input queue.	255
_POSIX_NAME_MAX	The number of bytes in a filename.	14
_POSIX_NGROUPS_MAX	The number of simultaneous supplementary group IDs per process.	0
_POSIX_OPEN_MAX	The number of files that one process can have open at one time.	16
_POSIX_PATH_MAX	The number of bytes in a pathame.	255
_POSIX_PIPE_BUF	The number of bytes that can be written atomically when writing to a pipe.	512

The following constants will always be defined in `<limits.h>`. They are invariant:

Name	Description	Minimum Acceptable Value
CHAR_BIT	Number of bits in a `char`	8
CHAR_MAX	Max integer value of a `char`	127
DBL_DIG	Digits of precision of a `double`	10
DBL_MAX	Max decimal value of a `double`	1E+37
FLT_DIG	Digits of precision of a `float`	6
FLT_MAX	Max decimal value of a `float`	1E+37
INT_MAX	Max decimal value of an `int`	32 767
LONG_BIT	Number of bits in a `long`	32
LONG_MAX	Max decimal value of a `long`	2 147 483 647
SCHAR_MAX	Max value of a signed `char`	127
SHRT_MAX	Max decimal value of a `short`	32 767
UCHAR_MAX	Max value of an `unsigned char`	255
UINT_MAX	Max value of an `unsigned int`	65 535
ULONG_MAX	Max value of an `unsigned long int`	4 294 967 295
USHRT_MAX	Max value for an `unsigned short int`	65 535
WORD_BIT	Number of bits in a ''word'' or `int`	16

Name	Description	Maximum Acceptable Value
CHAR_MIN	Min integer value of a `char`	0
DBL_MIN	Min decimal value of a `double`	1E–37
FLT_MIN	Min decimal value of a `float`	1E–37
INT_MIN	Min decimal value of a `int`	–32 768
LONG_MIN	Min decimal value of a `long`	–2 147 483 648
SCHAR_MIN	Min value of a signed `char`	–127
SHRT_MIN	Min decimal value of a `short`	–32 768

USAGE

If the value of an object of type `char` sign-extends when used in an expression, the value of `CHAR_MIN` is the same as that of `SCHAR_MIN` and the value of `CHAR_MAX` is the same as that of `SCHAR_MAX`. Otherwise, the value of `CHAR_MIN` is 0 and the value of `CHAR_MAX` will be the same as that of `UCHAR_MAX`.

SEE ALSO

fpathconf(BA_OS), sysconf(BA_OS).

LEVEL

Level 1.

NAME

locale: locale.h – category macros

SYNOPSIS

```
#include <locale.h>
```

DESCRIPTION

The <locale.h> header defines at least the following as macros:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
LC_MESSAGES
NULL
```

which expand to distinct integral-constant expressions, for use as the first argument to the setlocale() function [see setlocale(BA_OS)].

Declares the structure lconv which includes at least the following members:

```
char *decimal_point;       /* "." */
char *thousands_sep;       /* "" */
char *grouping;            /* "" */
char *int_curr_symbol;     /* "" */
char *currency_symbol;     /* "" */
char *mon_decimal_point;   /* "" */
char *mon_thousands_sep;   /* "" */
char *mon_grouping;        /* "" */
char *positive_sign;       /* "" */
char *negative_sign;       /* "" */
char int_frac_digits;      /* CHAR_MAX */
char frac_digits;          /* CHAR_MAX */
char p_cs_precedes;        /* CHAR_MAX */
char p_sep_by_space;       /* CHAR_MAX */
char n_cs_precedes;        /* CHAR_MAX */
char n_sep_by_space;       /* CHAR_MAX */
char p_sign_posn;          /* CHAR_MAX */
char n_sign_posn;          /* CHAR_MAX */
```

Declares setlocale() and localeconf() as a function.

Additional macro definitions, beginning with the characters LC_ and an upper case letter, may also be given here.

SEE ALSO

setlocale(BA_OS).

LEVEL

Level 1.

NAME

math: math.h – mathematical declarations

SYNOPSIS

```
#include <math.h>
```

DESCRIPTION

The <math.h> header provides for the following constants. The values are of type double and are accurate within the precision of the double type.

M_E	Value of e
M_LOG2E	Value of $\log_2 e$
M_LOG10E	Value of $\log_{10} e$
M_LN2	Value of $\log_e 2$
M_LN10	Value of $\log_e 10$
M_PI	Value of π
M_PI_2	Value of $\pi/2$
M_PI_4	Value of $\pi/4$
M_1_PI	Value of $1/\pi$
M_2_PI	Value of $2/\pi$
M_2_SQRTPI	Value of $2/\sqrt{\pi}$
M_SQRT2	Value of $\sqrt{2}$
M_SQRT1_2	Value of $1/\sqrt{2}$

The header contains a define statement for the MAXFLOAT symbol which is system dependent, and the value HUGE_VAL which is returned for error conditions found in the math library.

MAXFLOAT	Value of maximum non-infinite single-precision floating point number
HUGE_VAL	Error value returned by the math library

The macro HUGE_VAL is defined to represent error values returned by the math functions. HUGE_VAL will return either *+inf* on a system supporting IEEE Std 754-1985 or +{DBL_MAX} on a system that does not support the standard.

The following are declared as functions or macros:

```
acos()          cosh()          j0()            pow()
acosh()         erf()           j1()            scalb()
asin()          exp()           jn()            sin()
asinh()         fabs()          ldexp()         sinh()
atan2()         floor()         lgamma()        sqrt()
atan()          fmod()          log10()         tan()
atanh()         frexp()         log()           tanh()
cbrt()          isnan()         logb()          y0()
ceil()          gamma()         modf()          y1()
cos()           hypot()         nextafter()     yn()
```

Declares signgam as an external int.

SEE ALSO

Bessel(BA_LIB), erf(BA_LIB), exp(BA_LIB), floor(BA_LIB), frexp(BA_LIB), hyperbolic(BA_LIB), hypot(BA_LIB), lgamma(BA_LIB), trig(BA_LIB).

LEVEL

Level 1.

NAME

nl_types: nl_types.h – data types

SYNOPSIS

```
#include <nl_types.h>
```

DESCRIPTION

The `<nl_types.h>` header contains definitions of at least the following types:

`nl_catd` used by the message catalogue functions to identify a catalogue.

`nl_item` used by `nl_langinfo()` [see nl_langinfo(BA_LIB)] to identify items of langinfo data. Values of objects of type `nl_item` are defined in `<langinfo.h>` [see langinfo(BA_ENV)].

and at least the following constant:

`NL_SETD` used by the catalogue compiler when no `$set` directive is specified in a message text source file. This constant can be passed as the value of `set_id` on subsequent calls to `catgets()` [see catgets(BA_LIB)] (i.e., to retrieve messages from the default message set). The value of NL_SETD is implementation defined.

The following functions are declared:

```
catclose()
catgets()
catopen()
```

SEE ALSO

catopen(BA_LIB), catgets(BA_LIB), nl_langinfo(BA_LIB), langinfo(BA_ENV).

LEVEL

Level 1.

NAME

passwd – password file

SYNOPSIS

`/etc/passwd`

DESCRIPTION

The file `/etc/passwd` contains the following information for each user:

name
encrypted password (may be empty)
numerical user-ID
numerical group-ID (may be empty)
free field
initial-working-directory
program to use as command interpreter (may be empty)

This text file resides in directory `/etc`. It has general read permission and can be used, for example, to map *numerical user-ID*s to *name*s.

Each field within each user's entry is separated from the next by a colon. The field *encrypted password* may contain the encrypted password, nothing, or a lock string. The fields *numerical group-ID*, and *program to use as command interpreter* may be empty. However, if these fields are not empty, they must be used for their stated purpose. *free field* is a free field that is implementation-specific. Fields beyond the *program to use as command interpreter* field are also free but may be standardized in the future. Each user's entry is separated from the next by a newline.

The *name* is a character string that identifies a user.

By convention, the last element in the pathname of the initial-working-directory is typically *name*.

USAGE

In secure installations the `/etc/passwd` file may not contain the users actual password. Applications should not assume that the password in `/etc/passwd` is the user's actual password and should not use it for user authentication.

SEE ALSO

crypt(BA_LIB).

LEVEL

Level 1.

NAME

pwd: pwd.h – password structure

SYNOPSIS

```
#include <pwd.h>
```

DESCRIPTION

The <pwd.h> header provides a definition for `struct passwd`, which includes the following members:

```
char   *pw_name:         /* user's login name */
char   *pw_passwd;  /* encrypted password */
char   *pw_dir;          /* initial working directory */
char   *pw_shell;   /* program to use as shell */
```

The following are declared as either functions or macros:

```
getpwnam()   getpwuid()
```

SEE ALSO

getpwent(SD_LIB).

Level

Level 1.

NAME

regexp: regexp.h – regular-expression declarations

SYNOPSIS

```
#include <regexp.h>
```

DESCRIPTION

The `<regexp.h>` header declares the following functions as macros:

```
advance()   compile()   step()
```

and declares the following as external variables:

```
loc1   loc2   locs
```

SEE ALSO

regexp(BA_LIB).

FUTURE DIRECTIONS

The functionality of the regexp functions will eventually be replaced by a more complete interface and the regexp functions will be discontinued.

LEVEL

Level 2: September 30, 1989.

NAME

search: search.h – search tables

SYNOPSIS

```
#include <search.h>
```

DESCRIPTION

The <search.h> header provides a typedef, ENTRY, for struct entry which includes the following members:

```
char   *key;
char   *data;
```

and defines ACTION and VISIT as enumeration data types through typedefs as follows:

```
enum { FIND, ENTER } ACTION;
enum { preorder, postorder, endorder, leaf } VISIT;
```

The following are declared as either functions or macros:

```
hcreate()    lfind()     tdelete()
hdestroy()   lsearch()   tfind()
hsearch()    tsearch()   twalk()
```

SEE ALSO

hsearch(BA_LIB), lsearch(BA_LIB), tsearch(BA_LIB).

LEVEL

Level 1.

NAME

setjmp: setjmp.h – stack environment declarations

SYNOPSIS

```
#include <setjmp.h>
```

DESCRIPTION

The `<setjmp.h>` header contains the `typedef`s for types `jmp_buf` and `sigjmp_buf`.

The following are declared as functions: `longjmp()` and `siglongjmp()`.

Declares `setjmp()` and `sigsetjmp()` as either functions or macros.

SEE ALSO

setjmp(BA_LIB), sigsetjmp(BA_LIB).

LEVEL

Level 1.

NAME

siginfo – signal generation information

SYNOPSIS

```
#include <siginfo.h>
```

DESCRIPTION

If a process is catching a signal, it may request a record detailing why the system has generated that signal [see sigaction (BA_OS)]. If a process is monitoring its children, it may receive a record detailing the cause of any child's change of state [see waitid(BA_OS)]. In either case, the system will return that information in a structure of type `siginfo_t` that includes the following members:

```
int si_signo;   /* signal number */
int si_errno;   /* error number */
int si_code;    /* signal code */
```

`si_signo` contains the system generated signal number. (For the `waitid()` function, `si_signo` will always be equal to `SIGCHLD`.)

If `si_errno` is non-zero, it contains an error number associated with this signal, as defined in `errno.h`.

`si_code` contains a code identifying the cause of the signal. If the value of `si_code` is less than or equal to 0, then the signal was generated by a user process [see kill(BA_OS) and sigsend(BA_OS)] and the siginfo structure will contain the following additional members:

```
pid_t si_pid;    /* sending process ID */
uid_t si_uid;    /* sending user ID */
```

Otherwise, `si_code` contains a signal-specific reason why the signal was generated as follows:

Signal	Code	Reason
SIGILL	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
SIGFPE	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating point divide by zero
	FPE_FLTOVF	floating point overflow
	FPE_FLTUND	floating point underflow
	FPE_FLTRES	floating point inexact result

	FPE_FLTINV	invalid floating point operation
	FPE_FLTSUB	subscript out of range
SIGSEGV	SEGV_MAPERR	address not mapped to object
	SEGV_ACCERR	invalid permissions for mapped object
SIGBUS	BUS_ADRALN	invalid address alignment
	BUS_ADRERR	non-existent physical address
	BUS_OBJERR	object specific hardware error
SIGTRAP	TRAP_BRKPT	process breakpoint
	TRAP_TRACE	process trace trap
SIGCHLD	CLD_EXITED	child has exited
	CLD_KILLED	child was killed
	CLD_DUMPED	child has terminated abnormally
	CLD_TRAPPED	traced child has trapped
	CLD_STOPPED	child has stopped
	CLD_CONTINUED	stopped child has continued
SIGPOLL	POLL_IN	data input available
	POLL_OUT	output buffers available
	POLL_MSG	input message available
	POLL_ERR	I/O error
	POLL_PRI	high priority input available
	POLL_HUP	device disconnected

In addition, the following signal dependent information will be available:

Signal	Field	Value
SIGILL SIGFPE	caddr_t si_addr	address of faulting instruction
SIGSEGV SIGBUS	caddr_t si_addr	address of faulting memory reference
SIGCHLD	pid_t si_pid int si_status	child process ID exit value or signal
SIGPOLL	long si_band	band event for POLL_IN, POLL_OUT, or POLL_MSG

For some implementations, the exact value of `si_addr` may not be available; in that case, `si_addr` is guaranteed to be on the same page as the faulting instruction or memory reference.

SEE ALSO

kill(BA_OS), sigaction(BA_OS), signal(BA_ENV), sigsend(BA_OS), waitid(BA_OS).

LEVEL

Level 1.

NAME

signal – base signals

SYNOPSIS

```
#include <signal.h>
```

DESCRIPTION

The **<signal.h>** header defines the following data type through **typedef**:

sig_atomic_t Integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

and defines at least the following macros:

SIG_DFL
SIG_ERR
SIG_IGN

DESCRIPTION

A signal is an asynchronous notification of an event. A signal is said to be generated for (or sent to) a process when the event associated with that signal first occurs. Examples of such events include hardware faults, timer expiration and terminal activity, as well as the invocation of the **kill** or **sigsend** system calls. In some circumstances, the same event generates signals for multiple processes. The receiver may request a detailed notification of the source of the signal and the reason why it was generated [see **siginfo**(BA_ENV)].

Each process may a system action specified to be taken in response to each signal sent to it, called the signal's disposition. The set of system signal actions for a process is initialized from that of its parent. Once an action is installed for a specific signal, it usually remains installed until another disposition is explicitly requested by a call to either **sigaction, signal** or **sigset**, or until the process **exec**s [see **sigaction**(BA_OS) and **signal**(BA_OS)]. When a process **exec**s, all signals whose disposition has been set to catch the signal will be set to **SIG_DFL**. Alternatively, on request, the system will automatically reset the disposition of a signal to **SIG_DFL** after it has been caught [see **sigaction**(BA_OS) and **signal**(BA_OS)].

A signal is said to be delivered to a process when the appropriate action for the process and signal is taken. During the time between the generation of a signal and its delivery, the signal is said to be pending [see **sigpending**(BA_OS)]. Ordinarily, this interval cannot be detected by an application. However, a signal can be blocked from delivery [see **signal**(BA_OS) and **sigprocmask**(BA_OS)]. If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the process, the signal remains pending until either it is unblocked or the signal's disposition requests that the signal be ignored. If the signal disposition of a blocked signal requests that the signal be ignored, and if that signal is generated for the process, the signal is discarded immediately upon generation.

Each process has a signal mask that defines the set of signals currently blocked from delivery to it [see **sigprocmask**(BA_OS)]. The signal mask for a process is initialized from that of its creator.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated.

The signals currently defined in **sys/signal.h** are as follows:

Name	Default	Event
SIGHUP	Exit	Hangup [see **termio**(7)]
SIGINT	Exit	Interrupt [see **termio**(7)]
SIGQUIT	Core	Quit [see **termio**(7)]
SIGILL	Core	Illegal Instruction
SIGTRAP	Core	Trace/Breakpoint Trap
SIGABRT	Core	Abort
SIGEMT	Core	Emulation Trap
SIGFPE	Core	Arithmetic Exception
SIGKILL	Exit	Killed
SIGBUS	Core	Bus Error
SIGSEGV	Core	Segmentation Fault
SIGSYS	Core	Bad System Call
SIGPIPE	Exit	Broken Pipe
SIGALRM	Exit	Alarm Clock
SIGTERM	Exit	Terminated
SIGUSR1	Exit	User Signal 1
SIGUSR2	Exit	User Signal 2
SIGCHLD	Ignore	Child Status
SIGPWR	Ignore	Power Fail/Restart
SIGWINCH	Ignore	Window Size Change
SIGURG	Ignore	Urgent Socket Condition
SIGPOLL	Ignore	Socket I/O Possible
SIGSTOP	Stop	Stopped (signal)
SIGTSTP	Stop	Stopped (user) [see **termio**(7)]
SIGCONT	Ignore	Continued
SIGTTIN	Stop	Stopped (tty input) [see **termio**(7)]
SIGTTOU	Stop	Stopped (tty output) [see **termio**(7)]
SIGVTALRM	Exit	Virtual Timer Expired
SIGPROF	Exit	Profiling Timer Expired
SIGXCPU	Core	CPU time limit exceeded [see **getrlimit**(2)]
SIGXFSZ	Core	File size limit exceeded [see **getrlimit**(2)]

The **signal**, **sigset** or **sigaction** system calls, can be used to specify one of three dispositions for a signal: take the default action for the signal, ignore the signal, or catch the signal.

Default Action: SIG_DFL

A disposition of **SIG_DFL** specifies the default action. The default action for each signal is listed in the table above and is selected from the following:

Exit When it gets the signal, the receiving process is to be terminated with all the consequences outlined in **exit**(BA_OS).

Core When it gets the signal, the receiving process is to be terminated with all the consequences outlined in **exit**(BA_OS). In addition, a ''core image'' of the process is constructed in the current working directory.

Stop When it gets the signal, the receiving process is to stop.

Ignore When it gets the signal, the receiving process is to ignore it. This is identical to setting the disposition to **SIG_IGN**.

Note that to support compatibility for applications written before this functionality in System V, typical configurations have **init** ignore **SIGXCPU** and **SIGXFSZ**. Processes wanting to receive **SIGXCPU** and **SIGXFSZ** must explicitly set the disposition to **SIG_DFL**.

Ignore Signal: SIG_IGN

A disposition of **SIG_IGN** specifies that the signal is to be ignored.

Catch Signal: *function address*

A disposition that is a function address specifies that, when it gets the signal, the receiving process is to execute the signal handler at the specified address. Normally, the signal handler is passed the signal number as its only argument; if the disposition was set with the **sigaction** function however, additional arguments may be requested [see **sigaction**(BA_OS)]. When the signal handler returns, the receiving process resumes execution at the point it was interrupted, unless the signal handler makes other arrangements. If an invalid function address is specified, results are undefined.

If the disposition has been set with the **sigset** or **sigaction** function, the signal is automatically blocked by the system while the signal catcher is executing. If a **longjmp** [see **setjmp**(BA_LIBC)] is used to leave the signal catcher, then the signal must be explicitly unblocked by the user [see **signal**(BA_OS) and **sigprocmask**(BA_OS)].

If execution of the signal handler interrupts a blocked system call, the handler is executed and the interrupted system call returns a –1 to the calling process with **errno** set to **EINTR**. However, if the **SA_RESTART** flag is set the system call will be transparently restarted.

NOTICES

Signal Disposition

The dispositions of the **SIGKILL** and **SIGSTOP** signals cannot be altered from their default values. The system will generate an error if this is attempted.

The **SIGKILL** and **SIGSTOP** signals cannot be blocked. The system silently enforces this restriction.

Whenever a process receives a **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, or **SIGTTOU** signal, regardless of its disposition, any pending **SIGCONT** signal will be discarded. A process stopped by the above four signals is said to be in a job control stop.

Whenever a process receives a **SIGCONT** signal, regardless of its disposition, any pending **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU** signals will be discarded. In addition, if the process was stopped, it will be continued.

SIGPOLL is issued when a file descriptor corresponding to a **STREAMS** [see *BASE SYSTEM INTRODUCTION*] file has a "selectable" event pending. A process must specifically request that this signal be sent using the **I_SETSIG ioctl()** call. Otherwise, the process will never receive **SIGPOLL**.

If the disposition of the **SIGCHLD** signal has been set with the **signal()** or **sigset()** functions, or with the **sigaction()** function and the **SA_NOCLDSTOP** flag has been specified, it will only be sent to the calling process when its children exit; otherwise, it will also be sent when its children are stopped or continued due to job control.

If the signal occurs other than as the result of calling the **abort()** or **raise()** function, the behavior is undefined if the signal handler calls any function in the standard library, other than the **signal()** function itself, or refers to any object with static storage duration other than by assigning a value to a storage duration variable of type **volatile sig_atomic_t**.

When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this interface definition is unspecified if they are called from a signal-catching function. The following table defines a set of functions that are guaranteed to be either re-entrant or not interruptible by signals. Therefore applications may invoke them, without restriction, from signal-catching functions:

abort()	**fork()**	**read()**	**tcdrain()**
access()	**fstat()**	**rename()**	**tcflow()**
alarm()	**getegid()**	**rmdir()**	**tcflush()**
cfgetispeed()	**geteuid()**	**setgid()**	**tcgetattr()**
cfgetospeed()	**getgid()**	**setpgid()**	**tcgetpgrp()**
cfsetispeed()	**getgroups()**	**setsid()**	**tcsendbreak()**
cfsetospeed()	**getpgrp()**	**setuid()**	**tcsetattr()**
chdir()	**getpid()**	**sigaction()**	**tcsetpgrp()**
chmod()	**getppid()**	**sigaddset()**	**time()**
chown()	**getuid()**	**sigdelset()**	**times()**
chroot()	**kill()**	**sigemptyset()**	**umask()**
close()	**link()**	**sigfillset()**	**uname()**
creat()	**longjmp()**	**sigismember()**	**unlink()**
dup2()	**lseek()**	**signal()**	**ustat()**
dup()	**mkdir()**	**sigpending()**	**utime()**
execle()	**mkfifo()**	**sigprocmask()**	**wait()**
execve()	**open()**	**sigsuspend()**	**waitpid()**
_exit()	**pathconf()**	**sleep()**	**write()**
exit()	**pause()**	**stat()**	
fcntl()	**pipe()**	**sysconf()**	

All functions not in the above tables are considered to be unsafe with respect to signals. If any function that is unsafe is interrupted by a signal-catching function that then calls any function that is unsafe, the behavior is undefined.

The structure **sigaction** and the constants:

SA_ONSTACK
SA_RESETHAND
SA_RESTART
SA_SIGINFO
SA_NOCLDWAIT
SA_NOCLDSTOP

are defined for use with the function **sigaction()** [see sigaction(BA_OS)].

The constants:

SIG_BLOCK
SIG_UNBLOCK
SIG_SETMASK

are defined for use with the function **sigprocmask()** [see sigprocmask(BA_OS)].

The following are declared as functions or macros:

kill()	**sigemptyset()**	**sigpending()**
sigaction()	**sigfillset()**	**sigprocmask()**
sigaddset()	**sigismember()**	**sigsuspend()**
sigdelset()	**signal()**	

Considerations for Threads Programming

Signal disposition (that is, to default or to ignore or to trap by function a given signal type) is maintained at the process level and is shared by all threads. Signal masks, on the other hand, are maintained per thread.

Depending on circumstances (outlined below), caught signals are handled either by a specific thread or an arbitrary thread.

Synchronous Signals

Signals that are initiated by a specific thread (for example, division by zero, a request for a **SIGALRM** signal, a reference to an invalid address) are delivered to and handled by that thread. (Note: that thread will use the common handler function currently defined for the containing process.)

Asynchronous Signals

Signals that are not initiated by a specific thread (for example, a **SIGINT** signal from a terminal, a signal from another process via **kill**(BA_OS)) are handled by an arbitrary thread of the process that meets either of the following conditions.

The thread has a signal mask that *does not* include the type of the caught signal.

The thread is blocked is a **sigwait**(BA_OS) system call whose argument *does* include the type of the caught signal.

A caught signal will be delivered to only one thread of a process. Applications *cannot* predict which of several eligible threads will receive a caught signal. If this behavior is undesirable, applications should maintain only a single eligible thread per signal type.

Signal handling occurs only when a thread is scheduled to run. That latency can be reduced by having signals caught by (permanently) *bound* threads.

SEE ALSO

exit(BA_OS), getrlimit(BA_OS), kill(BA_OS), pause(BA_OS), raise(BA_OS), sigaction(BA_OS), sigalstack(BA_OS), siginfo(BA_ENV), signal(BA_OS), sigprocmask(BA_OS), sigsend(BA_OS), sigsetops(BA_OS), sigsuspend(BA_OS), streams(BA_DEV), termio(BA_DEV), wait(BA_OS).

LEVEL

Level 1.

NAME

stat: sys/stat.h – data returned by stat function

SYNOPSIS

```
#include <sys/stat.h>
```

DESCRIPTION

The <sys/stat.h> header defines the structure of the data returned by the functions stat() and fstat() [see stat(BA_OS)].

The structure stat contains at least the following members:

```
dev_t   st_dev;      /* ID of device containing file */
ino_t   st_ino;      /* file serial number */
mode_t  st_mode;     /* type of file (see below) */
nlink_t st_nlink;    /* number of links */
uid_t   st_uid;      /* user ID of file owner */
gid_t   st_gid;      /* group ID of file owner */
dev_t   st_rdev;     /* device ID (if file is character
                        or block special) */
off_t   st_size;     /* file size in bytes (if file is a
                        regular file) */
time_t  st_atime;    /* time of last access */
time_t  st_mtime;    /* time of last data modification */
time_t  st_ctime;    /* time of last status change */
long    st_blksize;  /* the preferred I/O block size for
                        this object */
long    st_blocks;   /* number of st_blksize blocks allocated
                        for this object */
```

The following symbolic names for the values of st_mode are also defined:

File type:

S_IFMT	type of file
S_IFBLK	block special
S_IFCHR	character special
S_IFDIR	directory
S_IFIFO	FIFO special
S_IFREG	regular
S_IFLNK	symbolic link

File modes:

S_IRWXU	read, write, execute/search by owner
S_IRUSR	read permission, owner
S_IWUSR	write permission, owner
S_IXUSR	execute/search permission, owner
S_IRWXG	read, write, execute/search by group
S_IRGRP	read permission, group
S_IWGRP	write permission, group

S_IXGRP	execute / search permission, group
S_IRWXO	read, write, execute / search by others
S_IROTH	read permission, others
S_IWOTH	write permission,others
S_IXOTH	execute / search permission, others
S_ISUID	set user ID on execution
S_ISGID	set group ID on execution
S_ISVTX	reserved

File type test macros:

S_ISBLK( )	test for a block special file
S_ISCHR( )	test for a character special file
S_ISDIR( )	test for a directory
S_ISFIFO( )	test for a FIFO special file
S_ISREG( )	test for a regular file

The following are declared as either functions or macros:

```
chmod()   mkfifo()
fstat()   mknod()
lstat()   stat()
mkdir()   umask()
```

USAGE

Use of the macros is recommended for determining the type of a file.

SEE ALSO

chmod(BA_OS), mkdir(BA_OS), mknod(BA_OS), stat(BA_OS), umask(BA_OS), types(BA_ENV).

LEVEL

Level 1.

NAME

stdarg: va_start, va_arg, va_end – handle variable argument list

SYNOPSIS

```
#include <stdarg.h>

void va_start(va_list ap, parmN);

type va_arg(va_list ap, type);

void va_end(va_list ap);
```

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists [see printf(BA_LIB)] but do not use the stdarg macros are inherently nonportable, because different machines use different argument-passing conventions.

`va_list` is a type defined for the variable used to traverse the list.

The `va_start()` macro is invoked before any access to the unnamed arguments and initializes *ap* for subsequent use by `va_arg()` and `va_end()`. The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `, ...`). If this parameter is declared with the `register` storage class or with a function or array type, the behavior is undefined.

The parameter *parmN* is required under strict ANSI C compilation. In other compilation modes, *parmN* need not be supplied and the second parameter to the `va_start()` macro can be left empty [*e.g.,* `va_start(ap, );`]. This allows for routines that contain no parameters before the `...` in the variable parameter list.

The `va_arg()` macro expands to an expression that has the type and value of the next argument in the call. The parameter *ap* should have been previously initialized by `va_start()`. Each invocation of `va_arg()` modifies *ap* so that the values of successive arguments are returned in turn. The parameter *type* is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The `va_end()` macro is used to clean up.

Multiple traversals, each bracketed by `va_start() ... va_end()`, are possible.

USAGE

It is up to the calling routine to specify how many arguments there are, because it is not always possible to determine this from the stack frame. For example, `execl()` is passed a zero pointer to signal the end of the list. `printf()` can tell how many arguments are there by the format. It is non-portable to specify a second argument of `char`, `short`, or `float` to `va_arg()`, because arguments seen by the called function are not `char`, `short`, or `float`. C converts `char` and `short` arguments to `int` and converts `float` arguments to `double` before passing them to a function.

EXAMPLE

The function `f1()` gathers into an array a list of arguments that are pointers to strings (but not more than `MAXARGS` arguments), then passes the array as a single argument to function `f2()`. The number of pointers is specified by the first argument to `f1()`.

```
#include <stdarg.h>
#define MAXARGS     31

void f1(int n_ptrs, ...)
{
     va_list ap;
     char *array[MAXARGS];
     int ptr_no = 0;

     if (n_ptrs > MAXARGS)
          n_ptrs = MAXARGS;
     va_start(ap, n_ptrs);
     while (ptr_no < n_ptrs)
          array[ptr_no++] = va_arg(ap, char *);
     va_end(ap);
     f2(n_ptrs, array);
}
```

Each call to `f1()` should have visible the definition of the function or a declaration such as

```
void f1(int, ...);
```

SEE ALSO

exec(BA_OS), printf(BA_LIB), vprintf(BA_LIB).

LEVEL

Level 1.

NAME

stddef: stddef.h – standard definitions

SYNOPSIS

```
#include <stddef.h>
```

DESCRIPTION

The following types and macros are defined in the standard header `<stddef.h>`. Some are also defined in other headers.

The types are:

`ptrdiff_t` signed integral type of the result of subtracting two pointers

`size_t` unsigned integral type of the result of the `sizeof` operator

`wchar_t` integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero. The space character, control characters representing horizontal tab, vertical tab and form feed, and each member of

```
[A-Za-z0-9!"#%&'()*+,-./:;<=>?[\]^_{|}~]
```

shall have a code value equal to its value when used as the lone character in an integer character constant.

The macros are `NULL` and

`offsetof(`*type*`,` *member-designator*`)`

which expands to an integral constant expression that has type `size_t`, the value of which is the offset, in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The *member-designator* shall be such that given

`static` *type* `t;`

then the expression `(t.`*member-designator*`)` evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

LEVEL

Level 1.

NAME

stdio: stdio.h – standard buffered input/output

SYNOPSIS

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

DESCRIPTION

The `<stdio.h>` header defines the following symbolic names:

`BUFSIZ`	Size of stdio buffers
`EOF`	End-of-file return value
`FILENAME_MAX`	Maximum size of character array to hold longest filename string
`FOPEN_MAX`	Maximum number of open streams
`_IOFBF`	Input/output fully buffered
`_IOLBF`	Input/output line buffered
`_IONBF`	Input/output unbuffered
`L_ctermid`	Maximum size of character array to hold `ctermid()` output
`L_cuserid`	Maximum size of character array to hold `cuserid()` output
`L_tmpnam`	Maximum size of character array to hold `tmpnam()` output
`NULL`	Null pointer
`P_tmpdir`	Path prefix used by `tmpnam()` and `tempnam()` for generated file names.
`SEEK_CUR`	Seek relative to current position
`SEEK_END`	Seek relative to end-of-file
`SEEK_SET`	Seek relative to start-of-file
`stderr`	Standard error output stream
`stdin`	Standard input stream
`stdout`	Standard output stream
`TMP_MAX`	Minimum number of unique filenames generated by `tmpnam()`
`NAME_MAX`	maximum number of characters in a filename

The following data type is defined through `typedef`:

`FILE`	A structure containing information about a file
`fpos_t`	An object type capable of recording all the information needed to specify uniquely every position within a file
`size_t`	Type returned by `sizeof` C-Language operator

The following are declared, as either functions or macros:

```
clearerr()      fscanf()        rename()
ctermid()       fseek()         rewind()
cuserid()       fsetpos()       scanf()
fclose()        ftell()         setbuf()
fdopen()        fwrite()        setvbuf()
feof()          getc()          sprintf()
ferror()        getchar()       sscanf()
fflush()        gets()          tempnam()
fgetc()         getw()          tmpfile()
fgetpos()       pclose()        tmpnam()
fgets()         perror()        ungetc()
fileno()        popen()         vfprintf()
fopen()         printf()        vprintf()
fprintf()       putc()          vsprintf()
fputc()         putchar()       putw()
fputs()         puts()          remove()
fread()         putw()
freopen()       remove()
```

SEE ALSO

ctermid(BA_LIB), cuserid(BA_OS), fclose(BA_OS), ferror(BA_OS), fopen(BA_OS), fread(BA_OS), fseek(BA_OS), getc(BA_LIB), getopt(BA_LIB), gets(BA_LIB), perror(BA_LIB), popen(BA_OS), printf(BA_LIB), putc(BA_LIB), puts(BA_LIB), rename(BA_OS), scanf(BA_LIB), setbuf(BA_LIB), stdio(BA_LIB), system(BA_OS), tmpfile(BA_LIB), tmpnam(BA_LIB), ungetc(BA_LIB), vprintf(BA_LIB).

LEVEL

Level 1.

NAME

stdlib: stdlib.h – standard library definitions

SYNOPSIS

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

DESCRIPTION

The `<stdlib.h>` header defines the following symbolic names:

`EXIT_FAILURE`	Unsuccessful termination
`EXIT_SUCCESS`	Successful termination
`MB_CUR_MAX`	Maximum number of bytes in a multibyte character for the extended character set specified by the current locale
`NULL`	null pointer
`RAND_MAX`	Maximum value returned by `rand()`

The following data type is defined through `typedef`:

`div_t`	Type returned by the `div()` function
`ldiv_t`	Type returned by the `ldiv()` function
`size_t`	Type returned by `sizeof` C-language operator
`wchar_t`	Type whose range can represent distinct codes for all members of the largest extended character set specified among supported locales

The following are declared as either functions or macros:

```
abort()     calloc()   malloc()     srand()
abs()       div()      mblen()      strtod()
atexit()    exit()     mbstowcs()   strtol()
atof()      free()     mbtowc()     strtoul()
atoi()      getenv()   qsort()      system()
atol()      labs()     rand()       wcstombs()
bsearch()   ldiv()     realloc()    wctomb()
```

SEE ALSO

bsearch(BA_LIB), malloc(BA_OS), qsort(BA_LIB), rand(BA_LIB), setlocale(BA_OS), strtod(BA_LIB).

LEVEL

Level 1.

NAME

string: string.h – string operations

SYNOPSIS

```
#include <string.h>
```

DESCRIPTION

The `<string.h>` header defines the following symbolic name:

`NULL` null pointer

and the following data type through `typedef`:

`size_t` Unsigned integral return of `sizeof` C-language operator.

The following are declared, as either functions or macros:

```
memccpy()       strcmp()        strncmp()
memchr()        strcoll()       strncpy()
memcmp()        strcpy()        strpbrk()
memcpy()        strcspn()       strrchr()
memmove()       strdup()        strspn()
memset()        strerror()      strstr()
strcat()        strlen()        strtok()
strchr()        strncat()       strxfrm()
```

SEE ALSO

memory(BA_LIB), string(BA_LIB), strcoll(BA_LIB), strerror(BA_LIB), strxfrm(BA_LIB).

LEVEL

Level 1.

NAME

tar: tar.h – extended tar definitions

SYNOPSIS

```
#include <tar.h>
```

DESCRIPTION

Header block definitions are:

General definitions:

Name	Description	Value
TMAGIC	"ustar"	ustar plus null byte
TMAGLEN	6	Length of the above
TVERSION	"00"	00 without a null byte
TVERSLEN	2	Length of the above

Typeflag field definitions:

Name	Description	Value
REGTYPE	'0'	Regular file
AREGTYPE	'\0'	Regular file
LNKTYPE	'1'	Link
SYMTYPE	'2'	Reserved
CHRTYPE	'3'	Character special
BLKTYPE	'4'	Block special
DIRTYPE	'5'	Directory
FIFOTYPE	'6'	FIFO special
CONTTYPE	'7'	Reserved

Mode field bit definitions (octal) :

Name	Description	Value
TSUID	04000	Set UID on execution
TSGID	02000	Set GID on execution
TSVTX	01000	Reserved
TUREAD	00400	Read by owner
TUWRITE	00200	Write by owner special
TUEXEC	00100	Execute/search by owner
TGREAD	00040	Read by group
TGWRITE	00020	Write by group
TGEXEC	00010	Execute/search by group
TOREAD	00004	Read by other
TOWRITE	00002	Write by other
TOEXEC	00001	Execute/search by other

SEE ALSO

tar(AU_CMD).

LEVEL

Level 1.

NAME

termios: termios.h – define values for termios

SYNOPSIS

```
#include <termios.h>
```

DESCRIPTION

The `<termios.h>` header contains the definitions used by the termios interfaces [see termios(BA_OS)].

Termios Structure

Unsigned integral type definitions exist for:

```
cc_t
speed_t
tcflag_t
```

The `termios` structure includes the following members:

```
tcflag_t  c_iflag;     /* input modes */
tcflag_t  c_oflag;     /* output modes */
tcflag_t  c_cflag;     /* control modes */
tcflag_t  c_lflag;     /* local modes */
cc_t      c_cc[NCCS];  /* control chars */
```

A definition is given for:

`NCCS` size of the array `c_cc` for control characters

The special control characters are defined by the array `c_cc`:

Subscript Usage		
Canonical Mode	Non-Canonical Mode	Description
VEOF		EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character
VSUSP	VSUSP	SUSP character
	VTIME	TIME character

The subscript values are unique, except that the `VMIN` and `VTIME` subscripts may have the same values as the `VEOF` and `VEOL` subscripts, respectively.

Input Modes

The `c_iflag` field describes the basic terminal input control:

BRKINT	Signal interrupt on break
ICRNL	Map CR to NL on input
IGNBRK	Ignore break condition
IGNCR	Ignore CR

IGNPAR	Ignore characters with parity errors
INLCR	Map NL to CR on input
INPCK	Enable input parity check
ISTRIP	Strip character
IUCLC	Map upper case to lower case on input
IXANY	Enable any character to restart output
IXOFF	Enable start/stop input control
IXON	Enable start/stop output control
PARMRK	Mark parity errors

Output Modes

The c_oflag field specifies the system treatment of output:

OPOST	Postprocess output
OLCUC	Map lower case to upper on output
ONLCR	Map NL to CR-NL on output
OCRNL	Map CR to NL on output
ONOCR	No CR output at column 0
ONLRET	NL performs CR function
OFILL	Use fill characters for delay
OFDEL	Fill is DEL, else NUL
NLDLY	Select newline delays:
NL0	Newline character type 0
NL1	Newline character type 1
CRDLY	Select carriage-return delays:
CR0	Carriage-return delay type 0
CR1	Carriage-return delay type 1
CR2	Carriage-return delay type 2
CR3	Carriage-return delay type 3
TABDLY	Select horizontal-tab delays:
TAB0	Horizontal-tab delay type 0
TAB1	Horizontal-tab delay type 1
TAB2	Horizontal-tab delay type 2
TAB3	Expand tabs to spaces
BSDLY	Select backspace delays:
BS0	Backspace-delay type 0
BS1	Backspace-delay type 1
VTDLY	Select vertical-tab delays:
VT0	Vertical-tab delay type 0
VT1	Vertical-tab delay type 1
FFDLY	Select form-feed delays:
FF0	Form-feed delay type 0
FF1	Form-feed delay type 1

Baud Rate Selection

The input and output baud rates are stored in the `termios` structure. These are the valid values for objects of type `speed_t`. The following values are defined, but not all baud rates need be supported by the underlying hardware.

`B0`	Hang up
`B50`	50 baud
`B75`	75 baud
`B110`	110 baud
`B134`	134.5 baud
`B150`	150 baud
`B200`	200 baud
`B300`	300 baud
`B600`	600 baud
`B1200`	1200 baud
`B1800`	1800 baud
`B2400`	2400 baud
`B4800`	4800 baud
`B9600`	9600 baud
`B19200`	19200 baud
`B38400`	38400 baud

Control Modes

The `c_cflag` field describes the hardware control of the terminal; not all values specified are required to be supported by the underlying hardware:

`CSIZE`	Character size:
`CS5`	5 bits
`CS6`	6 bits
`CS7`	7 bits
`CS8`	8 bits
`CSTOPB`	Send two stop bits, else one
`CREAD`	Enable receiver
`PARENB`	Parity enable
`PARODD`	Odd parity, else even
`HUPCL`	Hang up on last close
`CLOCAL`	Local line, else dial-up

Local Modes

The `c_lflag` field of the argument structure is used to control various terminal functions:

`ECHO`	Enable echo
`ECHOE`	Echo erase character as error-correcting backspace
`ECHOK`	Echo `KILL`
`ECHONL`	Echo `NL`
`ICANON`	Canonical input (erase and kill processing)
`IEXTEN`	Enable extended input character processing
`ISIG`	Enable signals
`NOFLSH`	Disable flush after interrupt or quit
`TOSTOP`	Send `SIGTTOU` for background output
`XCASE`	Canonical upper/lower presentation

Attribute Selection

The following symbolic constants for use with `tcsetattr()` [see `tcsetattr()` in termios(BA_OS)] are defined:

`TCSANOW`	change attributes immediately
`TCSADRAIN`	change attributes when output has drained
`TCSAFLUSH`	change attributes when output has drained; also flush pending input

Line Control

The following symbolic constants for use with `tcflush()` [see `tcflush()` in termios(BA_OS)] are defined:

`TCIFLUSH`	flush pending input
`TCOFLUSH`	flush untransmitted output
`TCIOFLUSH`	flush both pending input and untransmitted output

The following symbolic constants for use with `tcflow()` [see `tcflow()` in termios(BA_OS)] are defined:

`TCIOFF`	transmit a `STOP` character, intended to suspend input data
`TCION`	transmit a `START` character, intended to restart input data
`TCOOFF`	suspend output
`TCOON`	restart output

The following are declared as either functions or macros:

```
cfgetispeed()   tcflow()      tcsendbreak()
cfgetospeed()   tcflush()     tcsetattr()
cfsetispeed()   tcgetattr()   tcsetgrp()
cfsetospeed()   tcgetgrp()
tcdrain()       tcgetsid()
```

SEE ALSO

termios(BA_OS), termio(BA_DEV).

LEVEL

Level 1.

NAME

time: time.h – time types

SYNOPSIS

```
#include <time.h>
```

DESCRIPTION

The `<time.h>` header declares the structure `tm`, which includes at least the following members:

```
int  tm_sec;     /* seconds [0, 61] */
int  tm_min;     /* minutes [0, 59] */
int  tm_hour;    /* hour [0, 23] */
int  tm_mday;    /* day of month [1, 31] */
int  tm_mon;     /* month of year [0, 11] */
int  tm_year;    /* year since 1900 */
int  tm_wday;    /* day of week [0, 6] (Sunday = 0) */
int  tm_yday;    /* day of year [0, 365] */
int  tm_isdst;   /* daylight savings flag */
```

This header defines the following symbolic names:

`NULL`	null pointer
`CLK_TCK`	number of clock ticks per second
`CLOCKS_PER_SEC`	number of units per second returned by `clock()`

and the following data types through `typedef`:

`clock_t`	Arithmetic type capable of representing time in `CLOCKS_PER_SEC`
`size_t`	Unsigned integral return of `sizeof` operator
`time_t`	Arithmetic type capable of representing time in seconds

The value of `CLK_TCK` may be variable and it should not be assumed that `CLK_TCK` is a compile-time constant. The value of `CLK_TCK` is the same as the value of `sysconf(_SC_CLK_TCK)` [see sysconf(BA_OS)].

The following are declared as either functions or macros:

```
asctime()   difftime()   mktime()    time()
clock()     gmtime()     strftime()  tzset()
ctime()     localtime()
```

and the following are declared as variables:

```
daylight   timezone   tzname[]
```

SEE ALSO

clock(BA_LIB), ctime(BA_LIB), mktime(BA_LIB), strftime(BA_LIB), sysconf(BA_OS), time(BA_OS).

LEVEL

Level 1.

NAME

times: sys/times.h – process and child process times structure

SYNOPSIS

```
#include <sys/times.h>
```

DESCRIPTION

The `<sys/times.h>` header defines the structure returned by `times()` [see times(BA_OS)], `struct tms`, and includes the following members:

```
clock_t    tms_utime;   /* User CPU time */
clock_t    tms_stime;   /* System CPU time
clock_t    tms_cutime;  /* User CPU time of terminated
                           child processes */
clock_t    tms_cstime;  /* System CPU time of terminated
                           child processes */
```

The type `clock_t` is defined through a `typedef`.

Declares the following as a function:

```
times()
```

SEE ALSO

times(BA_OS).

LEVEL

Level 1.

NAME

types: sys/types.h – data types

SYNOPSIS

```
#include <sys/types.h>
```

DESCRIPTION

The <sys/types.h> header define data types and includes definitions for at least the following types:

clock_t	Used for system times in CLK_TCKs or CLOCKS_PER_SEC
dev_t	Used for device IDs
gid_t	Used for group IDs
ino_t	Used for file serial numbers
†key_t	Used for inter-process communication
mode_t	Used for some file attributes
nlink_t	Used for link counts
off_t	Used for file sizes
pid_t	Used for process IDs
size_t	Used for sizes of objects
ssize_t	Used for count of bytes or error indication
time_t	Used for time in seconds
uid_t	Used for user IDs

†All of the types except those marked above are defined as arithmetic types of an appropriate length. Additionally, size_t is unsigned, and pid_t is signed.

USAGE

The following names are commonly used as extensions to the above. They are therefore reserved and portable applications should not use them:

```
addr_t
caddr_t
```

LEVEL

Level 1.

NAME

ucontext – user context

SYNOPSIS

```
#include <ucontext.h>
```

DESCRIPTION

The `ucontext` structure defines the context of a thread of control within an executing process.

This structure includes at least the following members:

```
ucontext_t   *uc_link
sigset_t     uc_sigmask
stack_t      uc_stack
mcontext_t   uc_mcontext
```

`uc_link` is a pointer to the context that will be resumed when this context returns. If `uc_link` is equal to 0, then this context is the main context, and the process will exit when this context returns.

`uc_sigmask` defines the set of signals that are blocked when this context is active [see sigprocmask(BA_OS)].

`uc_stack` defines the stack used by this context [see sigaltstack(BA_OS)].

`uc_mcontext` contains the saved set of machine registers and any implementation specific context data. Portable applications should not modify or access `uc_mcontext`.

SEE ALSO

getcontext(BA_OS), sigaction(BA_OS), sigprocmask(BA_OS), sigaltstack(BA_OS).

LEVEL

Level 1.

NAME

ulimit: ulimit.h – ulimit commands

SYNOPSIS

```
#include <ulimit.h>
```

DESCRIPTION

The `<ulimit.h>` header defines the symbolic constants used in the `ulimit()` function [see ulimit(BA_OS)].

Symbolic constants:

`UL_GETFSIZE`	get maximum file size
`UL_SETFSIZE`	set maximum file size

Declares the following as either a function or a macro:

`ulimit()`

SEE ALSO

ulimit(BA_OS).

LEVEL

Level 2: September 30, 1989.

NAME

unistd: unistd.h – standard symbolic constants and structures

SYNOPSIS

```
#include <unistd.h>
```

DESCRIPTION

The <unistd.h> header defines the symbolic constants and structures which are referenced elsewhere in the System V Interface Definition and which are not already defined or declared in some other header. The contents of this header are shown below.

The following symbolic constants are defined for the access() function [see access(BA_OS)]:

R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute (search) permission
F_OK	Test for existence of file

The constants F_OK, R_OK, W_OK and X_OK and the expressions R_OK | W_OK, R_OK | X_OK and R_OK | W_OK | X_OK all have distinct values.

Declares the constant

NULL	null pointer

The following symbolic constants are defined for the lseek() [see lseek(BA_OS)] and fcntl() [see fcntl(BA_OS)] functions (they have distinct values):

SEEK_SET	Set file offset to *offset*
SEEK_CUR	Set file offset to current plus *offset*
SEEK_END	Set file offset to EOF plus *offset*

The following symbolic constants are defined (with fixed values):

_POSIX_VERSION	Integer value indicating version of the POSIX standard
_XOPEN_VERSION	integer value indicating version of the XPG to which system is compliant

The following symbolic constants are defined if that option is present:

`_POSIX_CHOWN_RESTRICTED`	the use of `chown()` is restricted to a process with appropriate privileges
`_POSIX_JOB_CONTROL`	implementation supports job control (will be defined on all compliant systems)
`_POSIX_NO_TRUNC`	pathname components longer than `{NAME_MAX}` generate an error
`_POSIX_SAVED_IDS`	causes the exec functions [see exec(BA_OS)] to save effective user and group (will be defined on all compliant systems)
`_POSIX_VDISABLE`	terminal special characters defined in `<termios.h>` [see termios(BA_ENV)] can be disabled using this character

The following symbolic constants are defined for `sysconf()` [see sysconf(BA_OS)]:

```
_SC_ARG_MAX
_SC_CHILD_MAX
_SC_CLK_TCK
_SC_JOB_CONTROL
_SC_NGROUPS_MAX
_SC_OPEN_MAX
_SC_PAGESIZE
_SC_PASS_MAX
_SC_SAVED_IDS
_SC_VERSION
_SC_XOPEN_VERSION
```

The following symbolic constants are defined for `pathconf()` [see fpathconf(BA_OS)]:

```
_PC_CHOWN_RESTRICTED
_PC_LINK_MAX
_PC_MAX_CANON
_PC_MAX_INPUT
_PC_NAME_MAX
_PC_NO_TRUNC
_PC_PATH_MAX
_PC_PIPE_BUF
_PC_VDISABLE
```

The following symbolic constants are defined for `confstr()` [see confstr(BA_OS)]:

```
_CS_SYSNAME
_CS_HOSTNAME
_CS_RELEASE
_CS_VERSION
_CS_MACHINE
_CS_ARCHITECTURE
_CS_HW_SERIAL
_CS_HW_PROVIDER
_CS_SPRC_DOMAIN
```

The following symbolic constants are defined for file streams:

`STDIN_FILENO`	File number of `stdin`. It is `0`.
`STDOUT_FILENO`	File number of `stout`. It is `1`.
`STDERR_FILENO`	File number of `stderr`. It is `2`.

The following are declared as either functions or macros:

```
access()    execv()       getpgrp()    rmdir()
alarm()     execve()      getpid()     setgid()
chdir()     execvp()      getppid()    setpgid()
chown()     _exit()       getuid()     setsid()
close()     fork()        isatty()     setuid()
ctermid()   fpathconf()   link()       sleep()
cuserid()   getcwd()      lseek()      sysconf()
dup2()      getegid()     pathconf()   tcgetpgrp()
dup()       geteuid()     pause()      tcsetpgrp()
execl()     getgid()      pipe()       ttyname()
execle()    getgroups()   read()       unlink()
execlp()    getlogin()    rename()     write()
```

USAGE

The following values for constants are defined for systems compliant to this issue of the System V Interface Definition:

```
_POSIX_VERSION   198808L
_XOPEN_VERSION        3
```

SEE ALSO

access(BA_OS), alarm(BA_OS), chdir(BA_OS), chown(BA_OS), close(BA_OS), ctermid(BA_LIB), cuserid(BA_OS), dup(BA_OS), exec(BA_OS), exit(BA_OS), fcntl(BA_OS), fork(BA_OS), fpathconf(BA_OS), getcwd(BA_OS), getgroups(BA_OS), getlogin(BA_LIB), getpid(BA_OS), getuid(BA_OS), kill(BA_OS), link(BA_OS), lseek(BA_OS), open(BA_OS), pause(BA_OS), pipe(BA_OS), read(BA_OS), rmdir(BA_OS), setpgid(BA_OS), setsid(BA_OS), setuid(BA_OS), sleep(BA_OS), sysconf(BA_OS), termios(BA_OS), termios(BA_ENV), ttyname(BA_LIB), unlink(BA_OS), utime(BA_OS), write(BA_OS), limits(BA_ENV).

LEVEL

Level 1.

NAME

utime: utime.h – access and modification times structure

SYNOPSIS

```
#include <utime.h>
```

DESCRIPTION

The `<utime.h>` header declares the structure `utimbuf`, which includes the following members:

```
time_t  actime;   /* access time */
time_t  modtime;  /* modification time */
```

The times are measured in seconds since the Epoch.

The type `time_t` is declared in `<sys/types.h>` [see types(BA_ENV)].

Declares the following as a function.

```
utime()
```

SEE ALSO

utime(BA_OS), types(BA_ENV).

LEVEL

Level 1.

NAME

utsname: sys/utsname.h – system name structure

SYNOPSIS

```
#include <sys/utsname.h>
```

DESCRIPTION

The `<sys/utsname.h>` header defines `struct utsname`, which includes the following members:

```
char  sysname[{SYS_NMLN}];    /* Name of this implementation of
                                 the operating system */
char  nodename[{SYS_NMLN}];   /* Name of this node within an
                                 implementation-specified
                                 communications network */
char  release[{SYS_NMLN}];    /* Current release level of this
                                  implementation */
char  version[{SYS_NMLN}];    /* Current version level of this
                                 release */
char  machine[{SYS_NMLN}];    /* Name of the hardware type that
                                 the system is running on */
```

The data stored in the character arrays is terminated by a null character.

Declares the following as a function:

```
uname()
```

SEE ALSO

uname(BA_OS).

LEVEL

Level 1.

NAME

wait: sys/wait.h – declarations for waiting

SYNOPSIS

```
#include <sys/wait.h>
```

DESCRIPTION

The `<sys/wait.h>` header defines the following symbolic constants for use with the `waitpid()` function [see wait(BA_OS)]:

`WNOHANG`	do not hang if no status is available, return immediately
`WUNTRACED`	report status of stopped child process

and the following macros for analysis of process status values:

`WEXITSTATUS()`	return exit status
`WIFEXITED()`	true if child exited normally
`WIFSIGNALED()`	true if child exited due to uncaught signal
`WIFSTOPPED()`	true if child is currently stopped
`WSTOPSIG()`	return signal number that caused process to stop
`WTERMSIG()`	return signal number that caused process to terminate

The following are declared as either functions or macros.

```
wait()   waitpid()   waitid()
```

SEE ALSO

wait(BA_OS).

LEVEL

Level 1.

NAME

wchar – extended wide character utilities

SYNOPSIS

```
#include <wchar.h>
```

DESCRIPTION

The **wchar.h** header defines the data types listed below through **typedef**s:

wchar_t Integral type whose range of values can represent distinct wide character codes for all members of the largest character set specified among the locales supported by the compilation environment: the null character has the code value zero and each member of the Portable Character Set has a code value equal to its value when used as the lone character in an integer character constant.

wuchar_t The unsigned version of **wchar_t**.

mbstate_t A type that can represent the state of the conversion between wide and multibyte characters.

wint_t An integral type that is able to store any valid wide character value and **WEOF**.

wctype_t A scalar type (pointer or integer) that can hold values which represent locale specific character classification categories.

size_t Unsigned integral type which is the result of the *sizeof* operator.

The following functions are declared by the **wchar** header:

```
int             iswascii(wint_t wc);
int             iswalnum(wint_t wc);
int             iswalpha(wint_t wc);
int             iswcntrl(wint_t wc);
int             iswdigit(wint_t wc);
int             iswgraph(wint_t wc);
int             iswlower(wint_t wc);
int             iswprint(wint_t wc);
int             iswpunct(wint_t wc);
int             iswspace(wint_t wc);
int             iswupper(wint_t wc);
int             iswxdigit(wint_t wc);
int             iswctype(wint_t wc, wctype_t prop);
int             fwprintf(FILE *stream, const wchar_t *format, ...);
int             fwscanf(FILE *stream, const wchar_t *format, ...);
int             wprintf(const wchar_t *format, ...);
int             wscanf(const wchar_t *format, ...);
int             swprintf(wchar_t *s, size_t n,
                   const wchar_t *format, ...);
int             swscanf(const wchar_t *s, const wchar_t *format,
                   ...);
int             vfwprintf(FILE *stream, const wchar_t *format,
                   va_list arg);
int             vfwscanf(FILE *stream, const wchar_t *format,
```

```
                va_list arg);
int             vwprintf(const wchar_t *format, va_list arg);
int             vwscanf(const wchar_t *format, va_list arg);
int             vswprintf(wchar_t *s, size_t n,
                va_list arg);
int             vswscanf(const wchar_t *s, const wchar_t *format,
                va_listarg);
int             wctob(wint_t c);
int             mbsinit(const mbstate_t *ps);
int             mbrlen(const char *s, size_t n, mbstate_t *ps);
int             mbrtowc(wchar_t *pwc, const char *s, size_t n,
                mbstate_t *ps);
int             wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
size_t          mbsrtowcs(wchar_t *dst, const char **src, size_t len,
                mbstate_t *ps);
size_t          wcsrtombs(char *dst, const wchar_t **src, size_t len,
                mbstate_t *ps);
wint_t          fgetwc(FILE *stream);
wchar_t         *fgetws(wchar_t *s, int n,FILE *stream);
wint_t          fputwc(wint_t c,FILE stream);
int             fputws(const wchar_t s, FILE *stream);
wint_t          getwc(FILE *stream);
wint_t          getwchar(void);
wint_t          putwc(wint_t c,FILE *stream);
wint_t          putwchar(wint_t c);
wint_t          towlower(wint_t wc);
wint_t          towupper(wint_t wc);
wint_t          ungetwc(wint_t c,FILE *stream);
wctype_t        wctype(const char *property);
wchar_t         *wcscat(wchar_t *ws1,const wchar_t *ws2);
wchar_t         *wcschr(const wchar_t *ws,wint_t wc);
int             wcscmp(const wchar_t *ws1,const wchar_t *ws2);
int             wcscoll(const wchar_t *ws1,const wchar_t *ws2);
wchar_t         *wcscpy(wchar_t *ws1,const wchar_t *ws2);
size_t          wcscspn(const wchar_t *ws1,const wchar_t *ws2);
size_t          wcfstime(wchar_t *wcs,size_t maxsize,
                   const wchar_t *fmt,const struct tm *timptr);
size_t          wcslen(const wchar_t *ws1);
wchar_t         *wcsncat(wchar_t *ws1,const wchar_t *ws2,size_t n);
int             wcsncmp(const wchar_t *ws1,const wchar_t *ws2,
                   size_t n);
wchar_t         *wcsncpy(wchar_t *ws1,const wchar_t *ws2,size_t n);
wchar_t         *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
wchar_t         *wcsrchr(const wchar_t *ws,wintr_t wc);
size_t          wcsspn(const wchar_t *ws1,const wchar_t *ws2);
double          wcstod(const wchar_t *nptr,wchar_t **endptr);
float           wcstof(const wchar_t *nptr,wchar_t **endptr);
long double     wcstold(const wchar_t *nptr,wchar_t **endptr);
wchar_t         *wcstok(wchar_t *ws1,const wchar_t *ws2,
```

```
                    wchar_t **savept);
long int            wcstol(const wchar_t *nptr,wchar_t **endptr,
                    int base);
unsigned long       wcstoul(const wchar_t *nptr,wchar_t **endptr,
                    int base);
wchar_t             *wcsstr(const wchar_t *ws1,const wchar_t *ws2);
int                 wcswidth(const wchar_t *pwcs, size_t n);
size_t              wcsxfrm(wchar_t *ws1, const wchar_t *ws2,
                    size_t n);
int                 wcwidth(wint_t);
```

wchar defines the following macro names:

WEOF Constant expression that is returned by some of the above functions to indicate end-of-file.

NULL Null pointer constant.

LEVEL

Level 1.

NOTICES

If the feature test macro **_XOPEN_SOURCE** is defined, the following are available:

```
wchar_t *wcstok(wchar_t ws1,const wchar_t *ws2);
wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);
size_t wcsftime(wchar_t *wcs, size_t maxsize,
     const char *fmt, const struct tm *timptr);
```

and all the symbols from **stdio.h**.

Base OS Service Routines

The following section contains the manual pages for the BA_OS service routines.

FINAL COPY
June 15, 1995
File: ba_os.cov
svid

NAME

abort – generate an abnormal termination signal

SYNOPSIS

```
#include <stdlib.h>

void abort (void);
```

DESCRIPTION

abort first closes all open files, **stdio** streams, directory streams and message catalogue descriptors, if possible, then causes the signal **SIGABRT** to be sent to the calling process.

USAGE

The signal sent by **abort()**, **SIGABRT**, should not be caught or ignored by applications. [see **sh**(BU_CMD)].

SEE ALSO

catopen(BA_LIB), **exit**(BA_OS), **kill**(BU_CMD), **sdb**(SD_CMD), **sh**(BU_CMD) **signal**(BA_OS), **sigaction**(BA_OS), **stdio**(BA_LIB)

LEVEL

Level 1.

NAME

access – determine accessibility of a file

SYNOPSIS

```
#include <unistd.h>
int access(const char *path, int amode);
```

DESCRIPTION

The function `access()` checks the accessibility of the file named by the pathname pointed to by the *path* argument, for the file access permissions indicated by *amode*, using the real user ID in place of the effective user ID, and the real group ID in place of the effective group ID.

The symbolic constants for the argument *amode* are defined by the `<unistd.h>` header file and are as follows:

Name	*Description*
`R_OK`	test for read permission.
`W_OK`	test for write permission.
`X_OK`	test for execute (search) permission.
`F_OK`	test for existence of file.

The argument *amode* is either the bitwise inclusive OR of one or more of the values of the symbolic constants for `R_OK`, `W_OK`, and `X_OK` or is the value of the symbolic constant `F_OK`.

RETURN VALUE

Upon successful completion, the function `access()` returns a value of `0`; otherwise, it returns a value of `–1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `access()` fails and sets `errno` to:

`ENOTDIR`	if a component of the path prefix is not a directory.
`ENOENT`	if the named file does not exist or the path argument points to an empty string.
`EACCES`	if a component of the path prefix denies search permission, or if the permission bits of the file mode do not permit the requested access.
`EROFS`	if write access is requested for a file on a read-only file system.
`ENAMETOOLONG`	if the length of a pathname exceeds `{PATH_MAX}`, or a pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
`ELOOP`	if too many symbolic links are encountered in translating the path.

SEE ALSO

chmod(BA_OS), stat(BA_OS).

FUTURE DIRECTIONS

`EINVAL` will be returned in `errno` if the argument *amode* is invalid.

LEVEL

Level 1.

NAME

adjtime – correct the time to synchronize the system clock

SYNOPSIS

```
#include <sys/time.h>

int adjtime(struct timeval *delta, struct timeval *olddelta);
```

DESCRIPTION

The function `adjtime()` adjusts the system's notion of the current time, as returned by `gettimeofday()`, advancing or retarding it by the amount of time specified in the `struct timeval` pointed to by *delta*.

The adjustment is effected by speeding up (if that amount of time is positive) or slowing down (if that amount of time is negative) the system's clock by some small percentage, generally a fraction of one percent. Thus, the time is always a monotonically-increasing function. A time correction from an earlier call to `adjtime()` may not be finished when `adjtime()` is called again. The second call to `adjtime()` cancels the first call to `adjtime()`. If *delta* is `0`, then *olddelta* returns the status of the effects of the previous `adjtime()` call and there is no effect upon time correction as a result of this call. If *olddelta* is not a null pointer, then the structure it points to will contain, upon return, the number of seconds and/or microseconds still to be corrected from the earlier call. If *olddelta* is a null pointer, the corresponding information will not be returned.

This call may be used in time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The adjustment value will be silently rounded to the resolution of the system clock.

RETURN VALUE

Upon successful completion, the function `adjtime()` returns a value of `0`; otherwise, it returns a value `-1` and sets `errno` to indicate an error.

ERRORS

Under the following condition, the function `adjtime()` fails and sets `errno` to:

`EPERM` if the process does not have the appropriate privilege.

SEE ALSO

date(BU_CMD), gettimeofday(RT_OS).

FUTURE DIRECTIONS

The functionality of `adjtime()` will be supported in the future, but the means of expressing terms will be changed to POSIX P1003.4–compatible types when that standard is available.

LEVEL

Level 2: September 30, 1989.

NAME

alarm – set process alarm clock

SYNOPSIS

```
#include <unistd.h>

unsigned alarm(unsigned sec);
```

DESCRIPTION

alarm instructs the alarm clock of the process to send the signal **SIGALRM** to the process after the number of real time seconds specified by *sec* have elapsed [see **signal**(BA_OS)].

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is canceled.

The **fork** routine sets the alarm clock of a new process to **0** [see fork(BA_OS)]. A process created by the **exec** family of routines inherits the time left on the old process's alarm clock.

Return Values

alarm returns the amount of time previously remaining in the alarm clock of the calling process.

SEE ALSO

exec(BA_OS), **fork**(BA_OS), **pause**(BA_OS), **signal**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

In multithreaded applications, the alarm signal is delivered to only the requesting thread, no other.

A thread cannot respond to a signal until it is scheduled for execution. For multiplexed threads, there may be a time lag between delivery of the signal and the time it is scheduled to run. For improved response, consider using *bound* threads.

NAME

atexit – add program termination routine

SYNOPSIS

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

DESCRIPTION

The function `atexit()` adds the function *func* to a list of functions to be called without arguments upon normal termination of the program. Normal termination occurs by either a call to `exit()` or a return from `main()`. At least 32 functions may be registered by `atexit()` and the functions will be called in the reverse order of their registration.

RETURN VALUE

Upon successful completion, the function `atexit()` returns a value of zero; otherwise, it returns a non-zero value.

SEE ALSO

exit(BA_OS).

LEVEL

Level 1.

NAME

chdir, fchdir – change working directory

SYNOPSIS

```
#include <unistd.h>

int chdir(const char *path);

int fchdir(int fildes);
```

DESCRIPTION

The functions `chdir()` and `fchdir()` cause a directory pointed to by *path* or referenced by the file descriptor *fildes* to become the current working directory, a directory that is the starting point for *path* searches of pathnames not beginning with slash.

For a directory to become the current working directory, a process must have execute (search) access to the directory. *path* points to the pathname of a directory. The *fildes* argument to `fchdir()` is a file descriptor of a directory obtained from a call to `open()` [see open(BA_OS)].

RETURN VALUE

Upon successful completion, the function `chdir()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error. On failure the current working directory remains unchanged.

ERRORS

Under the following conditions, the function `chdir()` fails and sets `errno` to:

`EACCES`	if search permission is denied for any component of the *path* name.
`ENOTDIR`	if a component of the pathname is not a directory.
`ENOENT`	if the named directory does not exist, or *path* points to an empty string.
`ELOOP`	if too many symbolic links were encountered in translating *path*.
`ENAMETOOLONG`	if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

Under the following conditions, the function `fchdir()` fails and sets `errno` to:

`EACCES`	if search permission is denied for *fildes*.
`EBADF`	if *fildes* is not an open file descriptor.
`ENOTDIR`	if the open file descriptor *fildes* does not refer to a directory.

SEE ALSO

chroot(KE_OS), open(BA_OS).

LEVEL

Level 1.

NAME

chmod, **fchmod** – change mode of file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);

int fchmod(int fildes, mode_t mode);
```

DESCRIPTION

chmod and **fchmod** set the access permission portion of the mode of the file whose name is given by *path* or referenced by the descriptor *fildes* to the bit pattern contained in *mode*. If *path* or *fildes* are symbolic links, the access permissions of the target of the symbolic links are set. Access permission bits are interpreted as follows:

S_ISUID	04000	Set user ID on execution.
S_ISGID	020#0	Set group ID on execution if # is **7**, **5**, **3**, or **1** Enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**
S_ISVTX	01000	Save text image after execution.
S_IRWXU	00700	Read, write, execute by owner.
S_IRUSR	00400	Read by owner.
S_IWUSR	00200	Write by owner.
S_IXUSR	00100	Execute (search if a directory) by owner.
S_IRWXG	00070	Read, write, execute by group.
S_IRGRP	00040	Read by group.
S_IWGRP	00020	Write by group.
S_IXGRP	00010	Execute by group.
S_IRWXO	00007	Read, write, execute (search) by others.
S_IROTH	00004	Read by others.
S_IWOTH	00002	Write by others
S_IXOTH	00001	Execute by others.

Modes are constructed by an OR of the access permission bits.

The effective user ID of the process must match the owner of the file or the process must have the appropriate privilege to change the mode of a file.

If the process does not have appropriate privilege and the file is not a directory, mode bit 01000 (save text image on execution) is cleared.

If the effective group ID of the process does not match the group ID of the file, and the process does not have appropriate privilege mode bit 02000 (set group ID on execution) is cleared.

If a 0410 executable file has the sticky bit (mode bit 01000) set, the operating system will not delete the program text from the swap area when the last user process terminates. If a 0413 or **ELF** executable file has the sticky bit set, the operating system will not delete the program text from memory when the last user process terminates. In either case, if the sticky bit is set the text will already be available (either in a swap area or in memory) when the next user of the file executes it, thus making execution faster.

If a directory is writable and the sticky bit, **S_ISVTX**, is set on the directory, a process may remove or rename files within that directory only if one or more of the following is true:

the effective user ID of the process is the same as that of the owner ID of the file

the effective user ID of the process is the same as that of the owner ID of the directory

the process has write permission for the file.

the process has appropriate privileges

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may affect future calls to **open**(BA_OS), **creat**(BA_OS), **read**(BA_OS), and **write**(BA_OS) on this file.

The following environment variables affect the execution of **chmod** [see envvar(BA_ENV)]:

LC_MESSAGES
Determines the locale to be used for diagnostic messages. If available, these messages will be retrieved from the message data base, **uxcore.abi**.

LC_ALL If a non-empty string, this overrides the values of all the other internationalization variables.

LANG The default value for internationalization variables that are unset or null.

Return Values

On success, **chmod** and **fchmod** return 0 and mark for update the **st_ctime** field of the file. On failure, **chmod** and **fchmod** return –1, set **errno** to identify the error, and the file mode is unchanged.

Errors

In the following conditions, **chmod** fails and sets **errno** to:

EACCES Search permission is denied on a component of the path prefix of *path*.

EACCES Write permission on the named file is denied.

EINTR A signal was caught during execution of the system call.

ELOOP Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG
The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.

ENOTDIR A component of the prefix of *path* is not a directory.

ENOENT Either a component of the path prefix, or the file referred to by *path* does not exist or is a null pathname.

EPERM The effective user ID does not match the owner of the file and the process does not have appropriate privilege (**P_OWNER**).

EROFS The file referred to by *path* resides on a read-only file system.

In the following conditions, **fchmod** fails and sets **errno** to:

EBADF *fildes* is not an open file descriptor

EINTR A signal was caught during execution of the **fchmod** system call.

ENOLINK *path* points to a remote machine and the link to that machine is no longer active.

EPERM The effective user ID does not match the owner of the file and the process does not have appropriate privilege (**P_OWNER**).

EROFS The file referred to by *fildes* resides on a read-only file system.

SEE ALSO

chown(BA_OS), **creat**(BA_OS), **exec**(BA_OS), **fcntl**(BA_OS), **mkfifo**(AS_CMD), **mknod**(BA_OS), **open**(BA_OS), **read**(BA_OS), **stat**(BA_OS), **write**(BA_OS)

LEVEL

Level 1.

The enforcement mode of file and record locking has moved to Level 2 effective September 30, 1989.

NAME

chown, **lchown**, **fchown** – change owner and group of a file

SYNOPSIS

```
#include <unistd.h>
#include <sys/stat.h>

int chown(const char *path, uid_t owner, gid_t group);

int lchown(const char *path, uid_t owner, gid_t group);

int fchown(int fildes, uid_t owner, gid_t group);
```

DESCRIPTION

The owner **ID** and group **ID** of the file specified by *path* or referenced by the descriptor *fildes*, are set to *owner* and *group* respectively. If *owner* or *group* is specified as –1, the corresponding **ID** of the file is not changed.

The function **lchown** sets the owner **ID** and group **ID** of the named file just as **chown** does, except in the case where the named file is a symbolic link. In this case **lchown** changes the ownership of the symbolic link file itself, while **chown** changes the ownership of the file or directory to which the symbolic link refers.

If **chown**, **lchown**, or **fchown** is invoked by a process without the **P_OWNER** privilege, the set-user-ID and set-group-ID bits of the file mode, **S_ISUID** and **S_ISGID** respectively, are cleared [see **chmod**(BA_OS)].

The operating system has a configuration option, {**_POSIX_CHOWN_RESTRICTED**}, that restricts ownership changes for the **chown**, **lchown**, and **fchown** system calls.

When {**_POSIX_CHOWN_RESTRICTED**} is not in effect, the effective user **ID** of the calling process must match the owner of the file or the process must have the **P_OWNER** privilege to change the ownership of a file.

When {**_POSIX_CHOWN_RESTRICTED**} is in effect, the **chown**, **lchown**, and **fchown** system calls prevent the owner of the file from changing the owner **ID** of the file and restrict the change of the group of the file to the list of supplementary group IDs. This restriction does not apply to calling processes with the **P_OWNER** privilege.

Return Values

On success, **chown**, **fchown** and **lchown** return 0 and mark for update the **st_ctime** field of the file. On failure, **chown**, **fchown** and **lchown** return –1, set **errno** to identify the error, and the owner and group of the file are unchanged.

Errors

In the following conditions, **chown** and **lchown** fail and set **errno** to:

EACCES Search permission is denied on a component of the path prefix of *path*.

EACCES Write permission on the named file is denied.

EINVAL *group* or *owner* is out of range.

ELOOP Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG
The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.

ENOTDIR A component of the path prefix of *path* is not a directory.

ENOENT Either a component of the path prefix or the file referred to by *path* does not exist or is a null pathname.

EPERM The effective user ID of the calling process does not match the owner of the file and the calling process does not have the appropriate privilege (**P_OWNER**) for changing file ownership.

EROFS The named file resides on a read-only file system.

In the following conditions, **fchown** fails and sets **errno** to:

EBADF *fildes* is not an open file descriptor.

EINVAL *group* or *owner* is out of range.

EPERM The effective user ID of the calling process does not match the owner of the file and the calling process does not have the appropriate privilege (**P_OWNER**) for changing file ownership.

EROFS The named file referred to by *fildes* resides on a read-only file system.

SEE ALSO

chgrp(AU_CMD), **chmod**(BA_OS), **chown**(AU_CMD)

LEVEL

Level 1.

NAME

close – close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fildes);
```

DESCRIPTION

close closes a file. *fildes* is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, **pipe**, or **iocntl** system call. **close** closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

Closing a file descriptor removes one reference to the associated file. When there are no more outstanding references to the file, if the link count of the file is zero, the space occupied by the file shall be freed and the file shall no longer be accessible.

If a STREAMS-based *fildes* is closed, and the calling process had previously registered to receive a **SIGPOLL** signal [see **signal**(BA_ENV)] for events associated with that stream [see **streams**(BA_DEV)], the calling process will be unregistered for events associated with the stream. The last **close** for a stream causes the stream associated with *fildes* to be dismantled. If **O_NONBLOCK** are clear and there have been no signals posted for the stream, and if there are data on the module's write queue, **close** waits up to 15 seconds (for each module and driver) for any output to drain before dismantling the stream. The time delay can be changed via an **I_SETCLTIME** **ioctl** request [see **streams**(BA_DEV)]. If **O_NONBLOCK** is set, or if there are any pending signals, **close** does not wait for output to drain, and dismantles the stream immediately.

If *fildes* is associated with one end of a pipe, the last **close** causes a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named [see **fattach**(BA_LIB)], the last **close** forces the named end to be detached [see **fdetach**(BA_LIB)]. If the named end has no open processes associated with it and becomes detached, the stream associated with that end is also dismantled.

Return Values

On success, **close** returns 0. On failure, **close** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **close** fails and sets **errno** to:

EBADF *fildes* is not a valid open file descriptor.

EINTR A signal was caught during the **close** system call.

SEE ALSO

creat(BA_OS), **dup**(BA_OS), **exec**(BA_OS), **fcntl**(BA_OS), **open**(BA_OS), **pipe**(BA_OS), **signal**(BA_OS), **signal**(BA_ENV), **streams**(BA_ENV)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Open file descriptors are global to the process and accessible to any sibling thread. If used concurrently, actions by one thread can interfere with those of a sibling.

A **close** executed by one thread will render the file descriptor unusable by all siblings. The **close** system call will block a thread that attempts to close a file descriptor that is in use (mid-system call) by a sibling.

NAME

confstr – obtain configurable string values

SYNOPSIS

```
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

DESCRIPTION

The **confstr** function provides a way for applications to obtain string values that are configuration-defined. There may be be similarities in terms of purpose and use with the **sysconf** function, although **confstr** is used with string return values rather than numeric return values. The argument *name* is the system variable that is being queried.

The **confstr** function provides the following valid values for *name*:

_CS_SYSNAME Copy the string that would be returned by **uname** [see **uname**(2)] in the *sysname* field, into the array pointed to by *buf*. This is the name of the implementation of the operating system, for example, **UNIX_SV**.

_CS_HOSTNAME
Copy a string that names the present host machine into the array pointed to by *buf*. This is the string that would be returned by **uname** in the *nodename* field. This hostname or nodename is often the name the machine is known by locally.

The *hostname* is the name of this machine as a node in some network; different networks may have different names for the node, but presenting the nodename to the appropriate network Directory or name-to-address mapping service should produce a transport end point address. The name may not be fully qualified.

Internet host names may be up to 256 bytes in length (plus the terminating null).

_CS_RELEASE Copy the string that would be returned by **uname** in the *release* field into the array pointed to by *buf*. Typical values might be **4.2**, **4.0**, **3.2**.

_CS_VERSION Copy the string that would be returned by **uname** in the *version* field into the array pointed to by *buf*. The syntax and semantics of this string are defined by the system provider.

_CS_MACHINE Copy the string that would be returned by **uname** in the *machine* field into the array pointed to by *buf*. For example, **i486**.

_CS_ARCHITECTURE
Copy a string describing the instruction set architecture of the current system into the array pointed to by *buf*. For example, **mc68030**, **i80486**. These names may not match predefined names in the C language compilation system.

_CS_HW_SERIAL
Copy a string which is the ASCII representation of the hardware-specific serial number of the physical machine on which the system call is executed into the array pointed to by *buf*. Note that this may be implemented in Read-Only Memory, via software constants set when building the operating system, or by other means, and may contain non-numeric characters. It is anticipated that manufacturers will not issue the same ''serial number'' to more than one physical machine. The pair of strings returned by **SI_HW_PROVIDER** and **SI_HW_SERIAL** is likely to be unique across operating system implementations.

_CS_HW_PROVIDER
Copies the name of the hardware manufacturer into the array pointed to by *buf*.

_CS_SRPC_DOMAIN
Copies the array pointed to by *buf* into the Secure Remote Procedure Call domain name.

The *name* value of **_CS_PATH**, defined in the **unistd.h** header, is supported by the implementation. Others may be supported. When *len* has a non-zero value and *name* has a value that is configuration-defined, *confstr* copies this value into the *len*-bytes buffer that *buf* is pointing to. If the string that is being returned is longer than *len* bytes, including the terminating null, the string is truncated to *len*-1 bytes by the **confstr** function. The result is also null-terminated.

To detect that the string has undergone a truncation process, the application makes a comparison between the value that the *confstr* function has returned and *len*. If *len* has the value zero and *buf* is a null pointer, an integer is still returned by **confstr**, as defined below, but it does not return a string. An unspecified result is produced if *len* is zero and *buf* is not a null pointer.

Return Values

If *name* has a value that is configuration-defined, **confstr** returns the size of the

The initial reason for having this function was to provide a way of finding the configuration-defined default value for the environment variable **PATH**. Applications need to be able to determine the system-supplied **PATH** environment variable value which contains the correct search paths for the various standard utilities. This is because **PATH** can be altered by users so that it can include directories that may contain utilities that replace standard utilities.

Examples

Here is an example of the use of **confstr** by an application:

```
confstr(name, (char *)NULL, (size_t)0);
```

In the example the **confstr** function is being used by the application to determine how big a buffer is needed for the string value. **malloc** could be used to allocate a buffer to hold the string. To obtain the string, **confstr** must be called again. An alternative is to allocate a fixed static buffer which is large enough to hold most answers, perhaps 512 or 1024 bytes. **malloc** could then be used to allocate a buffer that is larger in size if it finds that this is too small.

SEE ALSO

sysconf(BA_OS), **unistd**(BA_DEV)

LEVEL

Level 1.

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

DESCRIPTION

creat creates a new ordinary file or prepares to rewrite an existing file named by the pathname pointed to by *path.*

If the file exists, the length is truncated to 0 and the mode and owner are unchanged.

If the file does not exist the file's owner **ID** is set to the effective user **ID** of the process. The group **ID** of the file is set to the effective group **ID** of the process, or if the **S_ISGID** bit is set in the parent directory then the group **ID** of the file is inherited from the parent directory.

The mode bits of the file are based on the value of **mode**, modified as follows:

If the group **ID** of the new file does not match the effective group **ID** or one of the supplementary group IDs, the **S_ISGID** bit is cleared.

All bits set in the process file mode creation mask are cleared [see **umask**(2)].

The ''save text image after execution bit'' of the mode is cleared [see **chmod**(BA_OS) for the values of **mode**]

If **write** succeeds, it returns a write-only file descriptor and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across **exec** system calls [see **fcntl**(BA_OS)]. A new file may be created with a mode that forbids writing.

The call **creat(***path*, *mode***)** is equivalent to:

open(*path*, **O_WRONLY | O_CREAT | O_TRUNC,** *mode***)**

Return Values

On success, **creat** returns a non-negative integer, namely the lowest numbered unused file descriptor. On failure, **creat** returns –1, sets **errno** to identify the error, and no files are created or modified.

Errors

In the following conditions, **creat** fails and sets **errno** to:

EACCES Search permission is denied on a component of the path prefix.

EACCES

EAGAIN The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see **chmod**(BA_OS)].

EISDIR The named file is an existing directory.

EINTR A signal was caught during the **creat** system call.

ELOOP Too many symbolic links were encountered in translating *path*.

EMFILE The process has too many open files

ENAMETOOLONG
The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.

ENOTDIR A component of the path prefix is not a directory.

ENOENT A component of the path prefix does not exist.

ENOENT The pathname is null.

EROFS The named file resides or would reside on a read-only file system.

ENFILE The system file table is full.

ENOSPC The file system is out of inodes.

SEE ALSO

chmod(BA_OS), **close**(BA_OS), **fcntl**(BA_OS), **lseek**(BA_OS), **open**(BA_OS), **read**(BA_OS), **umask**(BA_OS), **write**(BA_OS)

LEVEL

Level 1.

The enforcement mode of file and record locking has moved to Level 2 effective September 30, 1989.

NOTICES

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

NAME

cuserid – get character login name of the user

SYNOPSIS

```
#include <unistd.h>
#include <stdio.h>

char *cuserid(char *s);
```

DESCRIPTION

The function `cuserid()` generates a character representation of the login name of the owner of the current process.

If *s* is a null pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least `L_cuserid` characters; the representation is left in this array. The constant `L_cuserid` is defined in the `<stdio.h>` header file, and has a value greater than zero.

RETURN VALUE

If the login name cannot be found, the function `cuserid()` returns a null pointer; if *s* is not a null pointer, a null character `(\0)` will be placed at *s*`[0]`.

SEE ALSO

getlogin(BA_LIB), getpwent(BA_LIB), logname(AU_CMD).

LEVEL

Level 1.

NAME

directory: **opendir**, **readdir**, **readdir_r**, **rewinddir**, **closedir** – directory operations

SYNOPSIS

```
#include <dirent.h>
#include <sys/types.h>
DIR *opendir(const char *filename);
struct dirent *readdir(DIR *dirp);
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
```

DESCRIPTION

opendir opens the directory named by *filename* and associates a directory stream with it. **opendir** returns a pointer to be used to identify the directory stream in subsequent operations. The directory stream is positioned at the first entry. A null pointer is returned if *filename* cannot be accessed or is not a directory, or if it cannot **malloc** enough memory to hold a **DIR** structure or a buffer for the directory entries.

readdir returns a pointer to the next active directory entry and positions the directory stream at the next entry. No inactive entries are returned. It returns **NULL** upon reaching the end of the directory or upon detecting an invalid location in the directory. **readdir** buffers several directory entries per actual read operation; **readdir** marks for update the **st_atime** field of the directory each time the directory is actually read. The structure **dirent** defined by the **<dirent.h>** header file describes a directory entry. It includes the filename (**d_name**), which is a null-terminated string of at most **{NAME_MAX}** characters:

```
char d_name[{NAME_MAX}]; /* name of file */
```

rewinddir resets the position of the named directory stream to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to **opendir** would.

closedir closes the named directory stream and frees the **DIR** structure.

Errors

The following errors can occur as a result of these operations.

opendir returns **NULL** on failure and sets **errno** to one of the following values:

ENOTDIR	A component of *filename* is not a directory.
EACCES	A component of *filename* denies search permission.
EACCES	Read permission is denied on the specified directory.
EMFILE	The maximum number of file descriptors are currently open.
ENFILE	The system file table is full.
ELOOP	Too many symbolic links were encountered in translating *filename*.

ENAMETOOLONG The length of the *filename* argument exceeds **{PATH_MAX}**, or the length of a *filename* component exceeds **{NAME_MAX}** while **{_POSIX_NO_TRUNC}** is in effect.

ENOENT A component of *filename* does not exist or is a null pathname.

readdir returns **NULL** on failure and sets **errno** to one of the following values:

ENOENT The current file pointer for the directory is not located at a valid entry.

EBADF The file descriptor determined by the **DIR** stream is no longer valid. This result occurs if the **DIR** stream has been closed.

closedir returns –1 on failure and sets **errno** to the following value:

EBADF The file descriptor determined by the **DIR** stream is no longer valid. This results if the **DIR** stream has been closed.

USAGE

Here is a sample program that prints the names of all the files in the current directory:

```
#include <stdio.h>
#include <dirent.h>

main()
{
     DIR *dirp;
     struct dirent *direntp;

     dirp = opendir(".");
     while ((direntp = readdir(dirp)) != NULL)
          (void)printf("%s\n", direntp->d_name);
     closedir(dirp);
     return (0);
}
```

SEE ALSO

dirent(BA_ENV), **mkdir**(BA_OS), **rmdir**(BA_OS)

LEVEL

Level 1.

NAME

dlclose – close a shared object

SYNOPSIS

```
#include <dlfcn.h>

int dlclose(void *handle);
```

DESCRIPTION

dlclose disassociates a shared object previously opened by **dlopen** from the current process. Once an object has been closed using **dlclose**, its symbols are no longer available to **dlsym**. All objects loaded automatically as a result of invoking **dlopen** on the referenced object [see **dlopen**(BA_OS)] are also closed. *handle* is the value returned by a previous invocation of **dlopen**.

Return Values

If the referenced object was successfully closed, **dlclose** returns 0. If the object could not be closed, or if *handle* does not refer to an open object, **dlclose** returns a non-0 value. More detailed diagnostic information is available through **dlerror**. If the system does not support dynamic linking of shared objects, **dlclose** returns -1 and sets **errno** to **ENOSYS**.

SEE ALSO

dlerror(BA_OS), **dlopen**(BA_OS), **dlsym**(BA_OS)

LEVEL

Level 1.

NAME

dlerror – get diagnostic information

SYNOPSIS

```
#include <dlfcn.h>

char *dlerror(void);
```

DESCRIPTION

dlerror returns a null-terminated character string (with no trailing newline) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of **dlerror**, **dlerror** returns **NULL**. Thus, invoking **dlerror** a second time, immediately following a prior invocation, results in **NULL** being returned.

Return Values

If the system does not support dynamic linking of shared objects, **dlerror** returns **NULL** and sets **errno** to **ENOSYS**.

SEE ALSO

dlclose(BA_OS), **dlopen**(BA_OS), **dlsym**(BA_OS)

LEVEL

Level 1.

NAME

dlopen – open a shared object

SYNOPSIS

```
#include <dlfcn.h>

void *dlopen(const char *pathname, int mode);
```

DESCRIPTION

Some implementations support the concept of a shared object. A shared object is an executable object file that another executable object file may load in constructing its own process image. A shared object may be loaded at different virtual addresses for different processes. A shared object may either be loaded when a process is created, if it was linked with the **a.out** form which the process was derived (see **ld**(SD_CMD)) or it may be loaded during the execution of the process.

dlopen makes a shared object available to a running process. **dlopen** returns to the process a *handle* the process may use on subsequent calls to **dlsym** and **dlclose**. This value should not be interpreted in any way by the process. *pathname* is the path name of the object to be opened; it may be an absolute path or relative to the current directory. If the value of *pathname* is 0, **dlopen** makes the symbols contained in the original **a.out**, all of the objects that were loaded at program startup with the **a.out**, and all objects loaded with the **RTLD_GLOBAL** mode, available through **dlsym**.

A shared object may specify other objects that it ''needs'' in order to execute properly. These needed objects are specified by special entries in the object file. Each needed object may, in turn, specify other needed objects. All such objects are loaded along with the original object as a result of the call to **dlopen**.

When a shared object is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. The *mode* parameter governs when these relocations take place and may have the following values:

RTLD_LAZY

Under this *mode*, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is invoked for the first time. This *mode* should result in better performance, since a process may not reference all of the functions in any given shared object.

RTLD_NOW

Under this *mode*, all necessary relocations are performed when the object is first loaded. This may result in some wasted effort, if relocations are performed for functions that are never referenced, but is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

Normally, a **dlopen**'d object's exported symbols are directly available only to those other objects that were loaded as a result of the same call to **dlopen**. If the *mode* argument is logically or'd with the value **RTLD_GLOBAL**, however, the exported symbols of all objects loaded via this call to **dlopen** are directly available to all other **dlopen**'d objects.

When searching for symbols to resolve a reference in one of the objects it is loading, the dynamic linker looks in the symbol tables of the objects it has already loaded. It uses the first occurence of the symbol that it finds. The first object searched is the **a.out**. Then come the **a.out**'s list of needed objects, in the order specified by the special entries in the **a.out**. Then come the second level list of needed entries, and so on. After all entries loaded on startup have been searched, the dynamic linker searches all objects loaded as the result of a call to **dlopen** (following the rules mentioned above for **RTLD_GLOBAL**). For each group, the object actually specified to **dlopen** is searched first, then that object's needed list, in order, then the second level needed entries, and so on. Since an object is loaded only once and may appear in the needed list of any number of objects, an object loaded with one call to **dlopen** or loaded on startup may be searched before the objects loaded for the current invocation of **dlopen**, even if it appears on the chain of dependencies for the object currently being **dlopen**'d.

Return Values

If *pathname* cannot be found, cannot be opened for reading, is not a shared object, or if an error occurs during the process of loading *pathname* or relocating its symbolic references, **dlopen** returns **NULL**. More detailed diagnostic information is available through **dlerror**. If the system does not support dynamic linking of shared objects, **dlopen** returns **NULL** and sets **errno** to **ENOSYS**.

SEE ALSO

dlclose(BA_OS), **dlerror**(BA_OS), **dlsym**(BA_OS),

LEVEL

Level 1.

NAME

dlsym – get the address of a symbol in shared object

SYNOPSIS

```
#include <dlfcn.h>

void *dlsym(void *handle, const char *name);
```

DESCRIPTION

dlsym allows a process to obtain the address of a symbol defined within a shared object previously opened by **dlopen**. *handle* is a value returned by a call to **dlopen**; the corresponding shared object must not have been closed using **dlclose**. *name* is the symbol's name as a character string. **dlsym** searches for the named symbol in all shared objects loaded automatically as a result of loading the object referenced by *handle* [see **dlopen**(BA_OS)].

Return Values

If *handle* does not refer to a valid object opened by **dlopen**, or if the named symbol cannot be found within any of the objects associated with *handle*, **dlsym** returns **NULL**. More detailed diagnostic information is available through **dlerror**. If the system does not support dynamic linking of shared objects, **dlsym** returns **NULL** and sets **errno** to **ENOSYS**.

SEE ALSO

dlclose(BA_OS) **dlerror**(BA_OS), **dlopen**(BA_OS),

LEVEL

Level 1.

NAME

dup – duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>

int dup(int fildes);
dup, dup2 – duplicate an open file descriptor
```

DESCRIPTION

dup duplicates an open file descriptor. *fildes* is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, **pipe**, or **ioctl** system call. **dup** returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read / write).

The new file descriptor is set to remain open across **exec** system calls [see **fcntl**(BA_OS)].

The file descriptor returned is the lowest one available. The **dup2** argument *fildes2* is set to refer to the same file as the **dup2** argument *fildes*. If *fildes2* already refers to an open file, not fildes, this file descriptor is first closed. If *fildes2* refers to *fildes*, or if *fildes* is not a valid open file descriptor, *fildes2* will not be closed first.

Return Values

On success, **dup** returns a non-negative integer, namely the file descriptor. On failure, **dup** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **dup** fails and sets **errno** to:

EBADF *fildes* is not a valid open file descriptor.

EINTR A signal was caught during the **dup** system call.

EMFILE The process has too many open files [see **getrlimit**(BA_OS)].

ENOLINK *fildes* is on a remote machine and the link to that machine is no longer active.

In addition, the function **dup2** may return one of the following errors:

EBADF if *fildes2* is negative or greater than or equal to **{OPEN_MAX}**.

EMFILE if no file descriptors above *fildes2* are available.

SEE ALSO

close(BA_OS), **creat**(BA_OS), **exec**(BA_OS), **fcntl**(BA_OS), **getrlimit**(BA_OS), **open**(BA_OS), **pipe**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

NAME

exec: **execl**, **execv**, **execle**, **execve**, **execlp**, **execvp** – execute a file

SYNOPSIS

```
#include <unistd.h>

int execl (const char *path, const char *arg0, ...,
     const char *argn, (char *)0);

int execv (const char *path, char *const *argv);

int execle (const char *path, const char *arg0, ...,
     const char *argn, (char *0), const char *envp[]);

int execve (const char *path, char *const *argv,
     char *const *envp);

int execlp (const char *file, const char *arg0, ...,
     const char *argn, (char *)0);

int execvp (const char *file, char *const *argv);
```

DESCRIPTION

exec in all its forms overlays a new process image on an old process. The new process image is constructed from an ordinary executable file. This file is either an executable object file or a file of data for an interpreter. There can be no return from a successful **exec** because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

#! *pathname* **[***arg***]**

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When you **exec** an interpreter file, the system **exec**s the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally executed file.

When a C program is executed, it is called as follows:

```
int main (int argc, char *argv[], char *envp[]);
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to null-terminated strings that constitute the environment for the new process. The value of the argument *argc* is conventionally at least one. The initial member of the array *argv* points to a string containing the name of the file.

The argument *path* points to a pathname that identifies the new process file. For **execlp** and **execvp**, the argument *file* points to the new process file. If the *file* argument does not contain a slash character, the path prefix for this file is obtained by searching the directories passed as the environment variable **PATH** [see **envvar**(BA_ENV) and **system**(BA_OS)]. The environment is supplied typically by the shell [see **sh**(BU_CMD)].

If the new executable file is not an executable object file, **execlp** and **execvp**

process ID
parent process ID
process group ID
supplementary group ID
semadj values
[see **semop**(KE_OS)]
session ID
[see **exit**(BA_OS) and **signal**(BA_OS)]
trace flag
[see **ptrace**(KE_OS) request 0]
time left until an alarm clock signal
[see **alarm**(BA_OS)]
current directory
root directory
file mode creation mask
[see **umask**(BA_OS)]
resource limits
[see **getrlimit**(BA_OS), **ulimit**(BA_OS)]
utime, **stime**, **cutime**, and **cstime**
[see **times**(BA_OS)].
file-locks
[see **fcntl**(BA_OS) and **lockf**(BA_OS)]
controlling terminal
process signal mask
[see **sigprocmask**(BA_OS)]
pending signals
[see sigpending(BA_OS)]

If **exec** succeeds, it marks for update the **st_atime** field of the file.

If **exec** succeeds, an internal reference to the process image file is created. This reference is removed some time later, but not later than process termination or successful completion of a subsequent call to one of the **exec** functions.

Return Values

On success, **exec** overlays the calling process image with the new process image and there is no return to the calling process. If **exec** fails while it can still return to the calling process, it returns –1 and sets **errno** to identify the error. If **exec** fails after a point of no return to the calling process, the calling process is sent a **SIGKILL** signal.

Errors

In the following conditions, **exec** fails and sets **errno** to:

EACCES	Search permission is denied for a directory listed in the new executable file's path prefix.
EACCES	The new executable file is not an ordinary file.
EACCES	Execute permission on the new executable file is denied.

E2BIG	The number of bytes in the argument list of the new process image is greater than the system-imposed limit of {**ARG_MAX**} bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.
ELOOP	Too many symbolic links were encountered in translating path or file.
ENAMETOOLONG	The length of the file or path argument exceeds **{PATH_MAX}**, or the length of a file or path component exceeds **{NAME_MAX}** while **_POSIX_NO_TRUNC** is in effect.
ENOENT	One or more components of the pathname of the executable file do not exist, or *path* or *file* points to an empty string.
ENOTDIR	A component of the pathname of the executable file is not a directory.
ENOEXEC	The **exec** is not an **execlp** or **execvp**, and the new executable file has the appropriate access permission but an invalid magic number in its header.
ENOMEM	The new process image requires more memory than allowed by **RLIMIT_VMEM**

USAGE

Two interfaces are available for these functions. The list (ell) versions **execl**, **execle**, and **execlp** are useful when a known file with known arguments is being called. The arguments are the character strings that include the filename and the arguments. The variable (**v**) versions: **execv**, **execve**, and **execvp** are useful when the number of arguments is unknown. The arguments include a filename and a vector of strings containing the arguments.

If possible, applications should use the **system** routine, which is easier to use and supplies more functions than the **fork** and exec routines.

SEE ALSO

alarm(BA_OS), **envvar**(BA_ENV), **exit**(BA_OS), **fcntl**(BA_OS), **fork**(BA_OS), **getrlimit**(BA_OS), **lockf**(BA_OS), **nice**(KE_OS), **priocntl**(KE_OS), **ps**(BU_CMD), **ptrace**(KE_OS), **semop**(KE_OS), **sh**(BU_CMD), **signal**(BA_ENV), **sigaction**(BA_OS), **sigpending**(BA_OS), **sigprocmask**(BA_OS), **system**(BA_OS), **times**(BA_OS), **ulimit**(BA_OS), **umask**(BA_OS)

LEVEL

Level 1.

NAME

exit, _exit – terminate process

SYNOPSIS

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

DESCRIPTION

The functions `exit()` and `_exit()` terminate the calling process. The function `exit()` may cause additional processing to be done before the process exits [see atexit(BA_OS) and fclose(BA_OS)]. All functions registered by the `atexit()` function are called, in the reverse order of the registration. The function `_exit()` does not do additional processing before exiting.

In addition, the following consequences will occur:

All of the file descriptors, directory streams and message catalogue descriptors are closed.

A `SIGCHLD` signal is sent to the calling process's parent process.

If the calling process's parent process is executing either `wait()`, `waitpid()`, or `waitid()` [see wait(BA_OS), `waitpid()` in wait(BA_OS), and waitid(BA_OS), respectively], and has not set its `SA_NOCLDWAIT` flag [see sigaction(BA_OS)], it is notified of the calling process's termination, the calling process's status is made available to it, and the lifetime of the calling process ends.

If the parent process is not executing either `wait()`, `waitpid()`, or `waitid()`, and has not set its `SA_NOCLDWAIT` flag, the calling process is transformed into a zombie process. The status of the child process will be made available to it when it subsequently executes a wait function. At that time, the lifetime of the calling process will end.

If the parent process has set its `SA_NOCLDWAIT` flag, the status will be discarded, and the lifetime of the calling process will end immediately.

The parent process ID of all of the calling process's child processes is set to the process ID of a special system process. That is, these processes are inherited by a special system process.

If the process is a controlling process, a `SIGHUP` signal is sent to each process in the foreground process group of the controlling terminal allocated to the calling process and the controlling terminal is deallocated.

If the exit of the calling process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, then a `SIGHUP` and `SIGCONT` signal will be sent to each member of that process group.

If the value of *status* is zero or `EXIT_SUCCESS`, an implementation defined form of the status `successful termination` is returned. If the value of *status* is `EXIT_FAILURE`, an implementation defined form of status `unsuccessful termination` is returned. Otherwise the status returned is implementation defined.

RETURN VALUE

The functions `exit()` and `_exit()` do not return values.

USAGE

Normally, applications should use `exit()` rather than `_exit()`.

SEE ALSO

atexit(BA_OS), catopen(BA_LIB), fclose(BA_OS), signal(BA_ENV), termios(BA_OS), wait(BA_OS), waitid(BA_OS).

LEVEL

Level 1.

NAME

fclose, fflush – close or flush a stdio-stream

SYNOPSIS

```
#include <stdio.h>

int fclose(FILE *strm);

int fflush(FILE *strm);
```

DESCRIPTION

The function `fclose()` causes any buffered data for *strm* to be written out, and the stdio-stream to be closed. If the underlying file is not already at `EOF`, and the file is one capable of seeking, the file pointer is adjusted so that the next operation on the open file pointer deals with the byte after the last one read from or written to the file being closed.

The function `fclose()` is performed automatically for all open files upon calling the `exit()` routine.

If *strm* points to an output stdio-stream or an update stdio-stream on which the most recent operation was not input, the function `fflush()` causes any buffered data for *strm* to be written to that file. Any unread data buffered in *strm* is discarded. The stdio-stream remains open. If *strm* is `NULL`, all open for writing stdio-streams are flushed.

The functions `fclose()` and `fflush()` mark for update the `st_ctime` and `st_mtime` fields of the underlying file, if the stream was writable, and if buffered data had not been written to the file yet.

RETURN VALUE

Upon successful completion, the functions `fclose()` and `fflush()` return a value of `0`; otherwise, they return `EOF` if an error is detected.

ERRORS

Under the following conditions, the functions `fclose()` and `fflush()` fail and set `errno` to:

`EAGAIN` if the `O_NONBLOCK` flag is set for the underlying file descriptor and the process would have blocked.

`EBADF` if the file descriptor underlying *strm* is not a valid file descriptor.

`EPIPE` if an attempt is made to write to a FIFO that is not open for reading by any process. A `SIGPIPE` signal is also sent to the process.

`EFBIG` if an attempt was made to write a file that exceeds the process's file size limit [see getrlimit(BA_OS)].

`EINTR` if a signal was caught during the `fclose()` or `fflush()` operation.

`ENOSPC` if there is no free space remaining on the device containing the file.

`EIO` if a physical I/O error has occurred, or if the process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU` and the process group of the process is orphaned.

SEE ALSO

close(BA_OS), exit(BA_OS), fopen(BA_OS), setbuf(BA_LIB), write(BA_OS).

LEVEL

Level 1.

NAME

fcntl – file control

SYNOPSIS

```
#include <sys/types.h>
#include <sys/fcntl.h>
#include <unistd.h>

int fcntl (int fildes, int cmd, . . .  /* arg */);
```

DESCRIPTION

fcntl provides for control over open files. *fildes* is an open file descriptor

fcntl may take a third argument, *arg*, whose data type, value and use depend upon the value of *cmd*. *cmd* specifies the operation to be performed by **fcntl** and may be one of the following:

F_DUPFD Return a new file descriptor with the following characteristics:

Lowest numbered available file descriptor greater than or equal to the integer value given as the third argument.

Same open file (or pipe) as the original file.

Same file pointer as the original file (that is, both file descriptors share one file pointer).

Same access mode (read, write, or read/write) as the original file.

Shares any locks associated with the original file descriptor.

Same file status flags (that is, both file descriptors share the same file status flags) as the original file.

The close-on-exec flag [see **F_GETFD**] associated with the new file descriptor is set to remain open across **exec**(BA_OS) system calls.

F_GETFD Get the close-on-exec flag associated with *fildes*. If the low-order bit is **0**, the file will remain open across **exec**. Otherwise, the file will be closed upon execution of **exec**.

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of the integer value given as the third argument (**0** or **1** as above).

F_GETFL Get *fildes* status flags.

F_SETFL Set *fildes* status flags to the integer value given as the third argument. Only certain flags can be set [see **fcntl**(BA_ENV)].

F_GETOWN Get the designated owner of the file.

F_SETOWN Set the owner field of the file descriptor.

F_FREESP Free storage space associated with a section of the ordinary file *fildes*. The section is specified by a variable of data type **struct flock** pointed to by the third argument *arg*. The data type **struct flock** is defined in the **sys/fcntl.h** header file and contains the following members: **l_whence** is 0, 1, or 2 to indicate that the relative offset **l_start** will be measured from the start of the file, the

current position, or the end of the file, respectively. **l_start** is the offset from the position specified in **l_whence**. **l_len** is the size of the section. An **l_len** of 0 frees up to the end of the file; in this case, the end of file (that is, file size) is set to the beginning of the section freed. Any data previously written into this section is no longer accessible.

The following commands are used for record-locking. Locks may be placed on an entire file or on segments of a file.

F_SETLK Set or clear a file segment lock according to the **flock** structure that *arg* points to The *cmd* **F_SETLK** is used to establish read (**F_RDLCK**) and write (**F_WRLCK**) locks, as well as remove either type of lock (**F_UNLCK**). If a read or write lock cannot be set, **fcntl** will return immediately with an error value of –1.

F_SETLKW This *cmd* is the same as **F_SETLK** except that if a read or write lock is blocked by other locks, **fcntl** will block until the segment is free to be locked.

F_GETLK Get the first lock which blocks the lock description pointed to by the third argument *arg*, taken as a pointer to the type **struct flock**. The information retrieved overwrites the information passed to **fcntl** in the structure **flock**. If no lock is found that would prevent this lock from being created, the structure is left unchanged except for the lock type which is set to **F_UNLCK**.

If the lock request described by the **flock** structure that *arg* points to could be created, then the structure is passed back unchanged except that the lock type is set to **F_UNLCK** and the **l_whence** field will be set to **SEEK_SET**.

This command never creates a lock; it tests whether a particular lock could be created.

F_RSETLK Used by the network lock daemon, to communicate with the NFS server kernel to handle locks on NFS files.

F_RSETLKW Used by the network lock daemon, to communicate with the NFS server kernel to handle locks on NFS files.

F_RGETLK Used by the network lock daemon, to communicate with the NFS server kernel to handle locks on NFS files.

F_RSETLK, **F_RSETLKW** and **F_RGETLK** are used by the **fslock** daemon and should not be used by regular applications.

A read lock prevents any other process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any other process from read locking or write locking the protected area. Only one write lock and no read locks may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The **flock** structure describes the type (**l_type**), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), process ID (**l_pid**), and system ID (**l_sysid**) of the segment of the file to be affected. The process ID and system ID fields are used only with the **F_GETLK** *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting **l_len** to **0**. If such a lock also has **l_whence** and **l_start** set to **0**, the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a **fork**(BA_OS) system call.

When mandatory file and record locking is active on a file [see **chmod**(BA_OS)], **creat**(BA_OS), **open**(BA_OS), **read**(BA_OS) and **write**(BA_OS) system calls issued on the file will be affected by the record locks in effect.

Return Values

On success, **fcntl** returns a value that depends on *cmd*:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined). The return value will not be negative.
F_SETFD	Value other than –1.
F_FREESP	Value of 0.
F_GETFL	Value of file status flags. The return value will not be negative.
F_SETFL	Value other than –1.
F_GETOWN	Value of the owner field.
F_SETOWN	Value other than –1.
F_GETLK	Value other than –1.
F_SETLK	Value other than –1.
F_SETLKW	Value other than –1.

On failure, **fcntl** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **fcntl** fails and sets **errno** to:

EACCES	*cmd* is **F_SETLK**, the type of lock (**l_type**) is a read lock (**F_RDLCK**) and the segment of a file to be locked is already write locked by another process, or the type is a write lock (**F_WRLCK**) and the segment of a file to be locked is already read or write locked by another process.

EACCES *cmd* is **F_SETFD, F_DETFL, F_SETLK, or F_SETLKW**, and either write permission on *fildes* is denied or *fildes* is already open for writing.

EACCES *cmd* is **F_SETLK** or **F_SETLKW**, mandatory file locking bit is set for the file, and the file is currently being mapped to virtual memory via **mmap** [see **mmap**(KE_OS)].

EAGAIN *cmd* is **F_FREESP**, the file exists, mandatory file/record locking is set, and there are outstanding record locks on the file.

EAGAIN *cmd* is **F_SETLK** or **F_SETLKW**, mandatory file locking bit is set for the file, and the file is currently being mapped to virtual memory via **mmap** [see **mmap**(KE_OS)].

EBADF *fildes* is not a valid open file descriptor.

EBADF *cmd* is **F_SETLK** or **F_SETLKW**, the type of lock (**l_type**) is a read lock (**F_RDLCK**), and *fildes* is not a valid file descriptor open for reading.

EBADF *cmd* is **F_SETLK** or **F_SETLKW**, the type of lock (**l_type**) is a write lock (**F_WRLCK**), and *fildes* is not a valid file descriptor open for writing.

EBADF *cmd* is **F_FREESP**, and *fildes* is not a valid file descriptor open for writing.

EDEADLK *cmd* is **F_SETLKW**, the lock is blocked by some lock from another process, and if **fcntl** blocked the calling process waiting for that lock to become free, a deadlock would occur.

EDEADLK *cmd* is **F_FREESP**, mandatory record locking is enabled, **O_NONBLOCK** is clear and a deadlock condition was detected.

EINTR A signal was caught during execution of the **fcntl** system call.

EIO

SEE ALSO

chown(BA_OS), **close**(BA_OS), **creat**(BA_OS), **exec**(BA_OS), **open**(BA_OS), **pipe**(BA_OS)

LEVEL

Level 1

The enforcement mode of file and record locking has moved to Level 2 effective September 30, 1989.

NOTICES

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

File and record locks are based on process ID; consequently, all siblings share locks. It is possible for a record lock placed by one thread to be overlaid with a lock by a sibling. Other mechanisms should be used to coordinate concurrent access by multiple threads.

A new command, **F_DUP2**, has been added. See description above.

NAME

ferror, feof, clearerr, fileno – stdio-stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int ferror(FILE *strm);

int feof(FILE *strm);

void clearerr(FILE *strm);

int fileno(FILE *strm);
```

DESCRIPTION

The function `ferror()` determines if an I/O error has occurred when reading from or writing to the file associated with the named stream.

The function `feof()` determines if `EOF` is detected when reading *strm*.

The function `clearerr()` resets both the error and `EOF` indicator on *strm*. The `EOF` indicator is reset when the file pointer associated with *strm* is repositioned, *e.g.*, by the `fseek()` or `rewind()` routines [see fseek(BA_OS) and `rewind()` in fseek(BA_OS), respectively], or can be reset with `clearerr()`.

The function `fileno()` gets the integer file descriptor associated with *strm* [see open(BA_OS)].

RETURN VALUE

The function `ferror()` will return non-zero when an I/O error has previously occurred reading from or writing to *strm*; otherwise, the function `ferror()` will return zero.

The function `feof()` will return non-zero when `EOF` has previously been detected reading *strm*; otherwise, the function `feof()` will return zero.

The function `fileno()` will return the integer file descriptor number associated with *strm*.

USAGE

The function `fileno()` returns a file descriptor that can be used with non-stdio routines, such as `write()` and `lseek()` routines, to manipulate the associated file, but these routines are not recommended for use by application-programs.

SEE ALSO

fseek(BA_OS), fopen(BA_OS), lseek(BA_OS), open(BA_OS), write(BA_OS).

LEVEL

Level 1.

NAME

fopen, freopen, fdopen – open a stdio-stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(const char *path, const char *type);

FILE *freopen(const char *path, const char *type,
     FILE *strm);

FILE *fdopen(int fildes, const char *type);
```

DESCRIPTION

The function `fopen()` opens the file named by *path* and associates a stdio-stream with it. The function `fopen()` returns a pointer to the `FILE` structure associated with the stdio-stream.

The function `freopen()` substitutes the named file in place of the open *strm*. A flush is first attempted and then the original *strm* is closed, regardless of whether the open ultimately succeeds. Failure to flush or close *strm* successfully is ignored. The function `freopen()` returns a pointer to the `FILE` structure associated with *strm*.

The function `freopen()` is typically used to attach the preopened stdio-streams associated with `stdin`, `stdout` and `stderr` to other files. The standard error output stream `stderr` is by default unbuffered but use of the function `freopen()` will cause it to become buffered or line-buffered.

The function `fdopen()` associates a stream with a file descriptor, *fildes*. The *type* of stream given to `fdopen()` must agree with the mode of the already open file. File-descriptors are obtained from routines which open files but do not return pointers to a `FILE` structure [`open()`, for example; see open(BA_OS)]. The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor. The error and `EOF` indicators for the stream are cleared. Streams are necessary input for many of the stdio routines.

The argument *path* points to a character-string that names the file to be opened.

The argument *type* is a character-string having one of the following values:

`r`	open text file for reading.
`w`	truncate to zero length or create text file for writing.
`a`	append; open for writing at the end of the text file, or create for writing.
`rb`	open binary file for reading.
`wb`	truncate to zero length or create binary file for writing.
`ab`	append; open or create binary file for writing at end-of-file.
`r+`	open text file for update (reading and writing).
`w+`	truncate or create text file for update.

`a+`	append; open or create text file for writing at end-of-file.
`r+b` or `rb+`	open binary file for update (reading and writing).
`w+b` or `wb+`	truncate or create binary file for update.
`a+b` or `ab+`	append; open or create binary file for writing at end-of-file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening call to the `fseek()`, `fflush()`, `rewind()` or `fsetpos()` routine [see fseek(BA_OS), `fflush()` in fclose(BA_OS), `rewind()` in fseek(BA_OS), and fsetpos(BA_OS), respectively]; and input may not be directly followed by output without an intervening call to the `fseek()`, `rewind()` or `fsetpos()` routine, unless the input operation encountered end-of-file.

If a file is opened for writing (*i.e.*, when *type* is `w`, `wb`, `w+` or `wb+`) and the file previously existed the `st_ctime` and `st_mtime` fields of the file will be updated. If a file is opened for writing or appending (*i.e.*, when *type* is `w`, `wb`, `w+` `wb+`, `a`, `ab`, `a+` or `ab+`) and the file did not previously exist, the `st_atime`, `st_ctime` and `st_mtime` fields of the file and the `st_ctime` and `st_mtime` fields of the parent directory will be updated.

When a file is opened for append (*i.e.*, when *type* is `a`, `ab`, `a+`, `a+b`, or `ab+`) it is impossible to overwrite information already in the file. The `fseek()` routine may be used to reposition the file-pointer to any position in the file, but when output is written to the file, the current file-pointer is disregarded. All output is written at the end of the file. For example, if two separate processes open the same file for append, each process may write to the file without overwriting output being written by the other, and the output from the two processes would be interleaved in the file.

When opened, a stdio-stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators are cleared for the stdio-stream.

RETURN VALUE

The functions `fopen()` and `freopen()` return a null pointer if *path* cannot be accessed, or if *type* is invalid, or if the file cannot be opened.

The function `fdopen()` returns a null pointer if *type* is invalid or if the file cannot be opened.

The functions `fopen()` or `fdopen()` may fail and not set `errno` if there are no free stdio streams.

ERRORS

Under the following conditions, the functions `fopen()` and `freopen()` fail and set `errno` to:

ENOTDIR if a component of the path-prefix in *path* is not a directory.

ENOENT if the named file does not exist or a component of the pathname should exist but does not, or *path* points to an empty string.

EACCES if a component of the path-prefix denies search permission, or *type* permission is denied for the named file, or the file does not exist and write permission is denied for the parent directory.

ELOOP if too many symbolic links are encountered in translating the path.

EISDIR if the named file is a directory and *type* is write or read / write.

ENAMETOOLONG
if the length of a pathname exceeds {PATH_MAX}, or pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

EINTR if a signal was caught during the open operation.

EMFILE if {OPEN_MAX} file descriptors are currently open in the calling process.

ENFILE if the system file table is full, meaning {SYS_OPEN} files are open in the system.

ENOSPC if the directory that would contain the file cannot be extended, the file does not exist, and it was to be created.

EROFS if the named file resides on a read-only file system and *type* requires write access.

ENXIO if the named file is a character special or block special file and the device associated with this special file does not exist.

USAGE

In System V, there is no difference between opening text and binary files, *i.e.*, opening a file with *type* "rb" is no different from opening a file with *type* "r".

SEE ALSO

creat(BA_OS), dup(BA_OS), fclose(BA_OS), fseek(BA_OS), open(BA_OS), pipe(BA_OS).

LEVEL

Level 1.

NAME

fork – create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

real user ID, real group ID, effective user ID, effective group ID
environment
close-on-exec flag [see **exec**(BA_OS)]
signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function address)
supplementary group IDs
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value [see **nice**(AS_CMD)]
scheduler class [see **priocntl**(RT_OS)]
all attached shared memory segments
process group **ID**
session **ID**
current working directory
root directory
file mode creation mask [see **umask**(BA_OS)]
resource limits
controlling terminal
working and maximum privilege sets

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class may or may not be inherited according to the policy of that particular class [see **priocntl**(RT_OS)].

The child process differs from the parent process in the following ways:

The child process has a unique process **ID** which does not match any active process group **ID**.

The child process has a different parent process **ID** (that is, the process **ID** of the parent process).

The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

All **semadj** values are cleared

Process locks, text locks and data locks are not inherited by the child

The child process's **tms** structure is cleared: **tms_utime**, **stime**, **cutime**, and **cstime** are set to 0

The time left until an alarm clock signal is reset to 0.

The set of signals pending for the child process is initialized to the empty set.

Record locks set by the parent process are not inherited by the child process [see **fcntl**(BA_OS)].

Return Values

On success, **fork** returns 0 to the child process and returns the process **ID** of the child process to the parent process. On failure, **fork** returns a value of **(pid_t)-1** to the parent process, sets **errno** to identify the error, and no child process is created.

Errors

In the following conditions, **fork** fails and sets **errno** to:

EAGAIN The system-imposed limit on the total number of processes under execution by a single user would be exceeded and the calling process does not have the **P_SYSOPS** privilege. The system lacked the necessary resources to create another process.

EAGAIN Total amount of system memory available when reading via raw I/O is temporarily insufficient.

SEE ALSO

exec (BA_OS), **fcntl** (BA_OS), **nice** (AS_CMD), **priocntl** (RT_OS), **signal** (BA_OS), **umask** (BA_OS), **wait** (BA_OS)

LEVEL

Level 1.

NAME

fpathconf, pathconf – get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>
long fpathconf(int fildes, int name);
long pathconf(const char *path
```

4. The behavior is undefined if *path* or *fildes* does not refer to a directory.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.
6. If *path* or *fildes* refers to a pipe or FIFO, the value returned applies to the FIFO itself. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fildes* refers to any other type of file, the behavior is undefined.
7. If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

The value of the configurable system limit or option specified by *name* will not change during the lifetime of the calling process.

RETURN VALUE

If the functions `fpathconf()` or `pathconf()` are invoked with an invalid symbolic constant, or if the symbolic constant corresponds to a configurable system limit or the option that is not supported on the system, a value of `-1` will be returned to the invoking process. If the function fails because the configurable system limit or option corresponding to *name* is not supported on the system the value of `errno` remains unchanged.

Otherwise, the functions `fpathconf()` and `pathconf()` return the current value for the file or directory.

ERRORS

Under the following conditions, the functions `fpathconf()` and `pathconf()` fail and set `errno` to:

`EINVAL`	if *name* is an invalid value.
`EINVAL`	if the implementation does not support an association of the variable *name* with the specified file.

The function `pathconf()` fails and sets `errno` to:

`EACCES`	if search permission is denied for a component of the path prefix
`ELOOP`	if too many symbolic links are encountered while translating *path*.
`ENAMETOOLONG`	if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
`ENOENT`	if *path* is needed for the command specified and the named file does not exist or if the *path* argument points to an empty string.
`ENOTDIR`	if a component of the path prefix is not a directory.

The function `fpathconf()` fails and sets `errno` to:

`EBADF` if the argument *fildes* is not a valid file descriptor.

SEE ALSO

sysconf(BA_OS).

LEVEL

Level 1.

NAME

fread, **fwrite** – binary input/output

SYNOPSIS

```
#include <stdio.h>

size_t fread (void *ptr, size_t size, size_t nitems, FILE *stream);

size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE
    *stream);
```

DESCRIPTION

fread reads into an array pointed to by *ptr* up to *nitems* items of data from *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. **fread** stops reading bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. **fread** increments the data pointer in *stream* to point to the byte following the last byte read if there is one. **fread** does not change the contents of *stream*. **fread** returns the number of items read.

fwrite writes to the named output *stream* at most *nitems* items of data from the array pointed to by *ptr*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. **fwrite** stops writing when it has written *nitems* items of data or if an error condition is encountered on *stream*. **fwrite** does not change the contents of the array pointed to by *ptr*. **fwrite** increments the data-pointer in *stream* by the number of bytes written. **fwrite** returns the number of items written.

If *size* or *nitems* is zero, then **fread** and **fwrite** return a value of 0 and do not effect the state of *stream*.

The **ferror** or **feof** routines must be used to distinguish between an error condition and end-of-file condition.

Return Values

On successful completion, the functions **fread** and **fwrite** return the number of items read or written, respectively. If *size* or *nitems* is non-positive, no characters are read or written, and both **fread** and **fwrite()** return a value of **0**. If an error occurs the error indicator for *strm* is set and **errno** is set to indicate the error.

Errors

If an error occurs, the error indicator for *stream* is set.

SEE ALSO

close (BA_OS), **open** (BA_OS), **getc** (BA_LIB), **gets** (BA_LIB), **lseek** (BA_OS), **printf** (BA_LIB), **putc** (BA_LIB), **puts** (BA_LIB), **read** (BA_OS), **scanf** (BA_LIB), **stdio** (BA_LIB), **write** (BA_OS)

LEVEL

Level 1.

NAME

fseek, rewind, ftell – reposition a file-pointer in a stdio-stream

SYNOPSIS

```
#include <stdio.h>

int fseek(FILE *strm, long int offset, int whence);

void rewind(FILE *strm);

long int ftell(FILE *strm);
```

DESCRIPTION

The function `fseek()` sets the position of the next input or output operation on *strm*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according to the value of *whence*, which is defined in the `<stdio.h>` header file as follows:

Name	*Description*
`SEEK_SET`	set position equal to *offset* bytes.
`SEEK_CUR`	set position to current location plus *offset*.
`SEEK_END`	set position to `EOF` plus *offset*.

The function `fseek()` allows the file position indicator to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return zero until data is actually written into the gap. The function `fseek()`, by itself, does not extend the size of the file. The behavior of `fseek()` on devices incapable of seeking is implementation defined.

The call `rewind(`*strm*`)` is equivalent to the following:

```
(void)fseek(strm, 0L, SEEK_SET)
```

except that the function `rewind()` clears the error indicator on *strm*.

The functions `fseek()` and `rewind()` clear the end-of-file indicator for *strm* and undo any effects of the `ungetc()` routine on the same stream. After `fseek()` or `rewind()`, the next operation on a file opened for update may be either input or output.

The function `ftell()` returns the offset of the current byte relative to the beginning of the file associated with *strm*. The offset is always measured in bytes.

If *strm* is writable and buffered data had not been written to the underlying file, the function `fseek()` will cause the unwritten data to be written to the file and mark the `st_ctime` and `st_mtime` fields of the file for update.

RETURN VALUE

Upon successful completion, the function `fseek()` returns a value of `0`. For improper seeks, it returns a value of `-1` and sets `errno` to indicate an error. An improper seek is, for example, an `fseek()` on a file that has not been opened via the `fopen()` routine or on a stream opened via the `popen()` routine.

Upon failure, the function `ftell()` returns a value of `–1` and sets `errno` to indicate an error.

ERRORS

Under the following condition, the functions `fseek()`, `rewind()` and `ftell()` fail and set `errno` to:

`EBADF` if the file descriptor underlying the stdio-stream is incorrect.

`EFBIG`

NAME

fsetpos, fgetpos – reposition a file pointer in a stdio-stream

SYNOPSIS

```
#include <stdio.h>

int fsetpos(FILE *strm, const fpos_t *pos);

int fgetpos(FILE *strm, fpos_t *pos);
```

DESCRIPTION

The function `fsetpos()` sets the position of the next input or output operation on *strm* according to the value of the node pointed to by *pos*. The node pointed to by *pos* must be a value returned by an earlier call to `fgetpos()` on the same stdio-stream.

`fsetpos()`

NAME

fsync – synchronize a file's in-memory state with that on the physical medium

SYNOPSIS

```
int fsync(int fildes);
```

DESCRIPTION

The function `fsync()` moves all modified data and attributes of *fildes* to a storage device; all in-memory modified copies of buffers for the associated file will have been written to the physical medium when the call returns. Note that this is different from `sync()`, which schedules disk I/O for all files but returns before the I/O completes. `fsync()` should be used by programs that require a file to be in a known state; for example, a program that contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction were recorded on the storage medium.

The way the data reaches the physical medium is implementation- and hardware-dependent. `fsync()` returns when the device driver tells it that the write has taken place.

RETURN VALUE

Upon successful completion, the function `fsync()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `fsync()` fails and sets `errno` to:

`EBADF`	if *fildes* is not a valid file descriptor open for writing.
`EINTR`	if a signal was caught during execution of the system call.
`EINVAL`	if the *fildes* argument does not refer to a file on which this operation is possible.
`EIO`	if an I/O error occurred while reading from or writing to the file system.

SEE ALSO

sync(BA_OS).

LEVEL

Level 1.

NAME

getcontext, **setcontext** – get and set current user context

SYNOPSIS

```
#include <ucontext.h>

int getcontext(ucontext_t *ucp);

int setcontext(ucontext_t *ucp);
```

DESCRIPTION

These functions, along with those defined in are useful for implementing user level context switching between multiple threads of control within a process.

getcontext initializes the structure pointed to by *ucp* to the current user context of the calling process. The user context is defined by and includes the contents of the calling process's machine registers, signal mask and execution stack.

setcontext restores the user context pointed to by *ucp*. The call to **setcontext** does not return; program execution resumes at the point specified by the context structure passed to **setcontext**. The context structure should have been one created either by a prior call to **getcontext** or **makecontext** or passed as the third argument to a signal handler [see **sigaction**(BA_OS)]. If the context structure was one created with **getcontext**, program execution continues as if the corresponding call of **getcontext** had just returned. If the context structure was one created with **makecontext**, program execution continues with the function specified to **makecontext**.

Return Values

On success, **setcontext** does not return and **getcontext** returns 0. On failure, **setcontext** and **getcontext** return –1 and set **errno** to identify the error.

SEE ALSO

setjmp(BA_LIB), **sigaction**(BA_OS), **sigprocmask**(BA_OS)

LEVEL

Level 1.

NOTICES

When a signal handler is executed, the current user context is saved and a new context is created by the kernel. If the process leaves the signal handler via **longjmp** [see **setjmp**(BA_LIB)] the original context will not be restored, and future calls to **getcontext** will not be reliable. Signal handlers should use **siglongjmp** [see **setjmp**(BA_LIB)] or **setcontext** instead.

NAME

getcwd – get pathname of current working directory

SYNOPSIS

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

DESCRIPTION

The function `getcwd()` places an absolute pathname of the current working directory in the array pointed to by *buf*. The value of *size* is the size in bytes of *buf*.

RETURN VALUE

Upon successful completion, the function `getcwd()` returns a pointer to the string containing the absolute pathname of the current working directory. Otherwise, the function `getcwd()` returns `NULL` if *size* is not large enough, or if an error occurs in a lower-level function.

ERRORS

Under the following conditions, the function `getcwd()` fails and sets `errno` to:

`EACCES` if a parent directory cannot be read to get its name.

`EINVAL` if *size* is less than or equal to zero.

`ERANGE` if *size* is greater than zero and less than the length of the pathname, plus `1`.

LEVEL

Level 1.

NAME

getgroups, setgroups – get or set supplementary group IDs

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

int getgroups(int gidsetsize, gid_t *grouplist);

int setgroups(int ngroups, const gid_t *grouplist);
```

DESCRIPTION

The `getgroups()` function fills in the array grouplist with the current supplementary group IDs of the calling process. The *gidsetsize* argument specifies the number of elements in the array grouplist and must be less than `{NGROUPS_MAX}`. The actual number of supplementary group IDs is returned. If *gidsetsize* is zero, `getgroups()` returns the number of supplementary group IDs associated with the calling process without modifying *grouplist*.

The function `setgroups()` sets the supplementary group access list of the calling process from the array of group IDs specified by *grouplist*. The number of entries is specified by *ngroups* and cannot be greater than `{NGROUPS_MAX}`. This function may be invoked only by a user with appropriate privileges.

RETURN VALUE

Upon successful completion, the function `getgroups()` returns the number of supplementary group IDs set for the calling process; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

The function `setgroups()` returns the value `0` upon successful completion. Otherwise, a value of `-1` is returned and `errno` is set to indicate an error.

ERRORS

Under the following condition, the function `getgroups()` fails and sets `errno` to:

`EINVAL` if the value of *gidsetsize* is non-zero and is less than the number of supplementary group IDs set for the calling process.

The function `setgroups()` fails and sets `errno` to:

`EINVAL` if the value of *ngroups* is greater than `{NGROUPS_MAX}`.

`EPERM` if the effective user ID is not that of a user with appropriate privileges.

SEE ALSO

chmod(BA_OS), getuid(BA_OS), initgroups(BA_LIB), setuid(BA_OS).

LEVEL

Level 1.

NAME

getmsg, getpmsg – get next message off a stream

SYNOPSIS

```
#include <stropts.h>

int getmsg(int fd, struct strbuf *ctlptr,
     struct strbuf *dataptr, int *flagsp);

int getpmsg(int fd, struct strbuf *ctlptr,
     struct strbuf *dataptr, int *bandp, int *flagsp);
```

DESCRIPTION

getmsg retrieves the contents of a message located at the stream head read queue from a STREAMS file, and places the contents into user specified buffer(s). The message must contain either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

The function **getpmsg** does the same thing as **getmsg**, but provides finer control over the priority of the messages received. Except where noted, all information pertaining to **getmsg** also pertains to **getpmsg**.

fd specifies a file descriptor referencing an open stream. *ctlptr* and *dataptr* each point to a **strbuf** structure, which contains the following members:

```
int maxlen;     /* maximum buffer length */
int len;        /* length of data */
char *buf;      /* ptr to buffer */
```

buf points to a buffer in which the data or control information is to be placed, and **maxlen** indicates the maximum number of bytes this buffer can hold. On return, **len** contains the number of bytes of data or control information actually received, or 0 if there is a zero-length control or data part, or -1 if no data or control information is present in the message. *flagsp* should point to an integer that indicates the type of message the user is able to receive. This is described later.

ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is **NULL** or the **maxlen** field is –1, the control (or data) part of the message is not processed and is left on the stream head read queue. If *ctlptr* (or *dataptr*) is not **NULL** and there is no corresponding control (or data) part of the messages on the stream head read queue, **len** is set to –1. If the **maxlen** field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and **len** is set to 0. If the **maxlen** field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and **len** is set to 0. If the **maxlen** field in *ctlptr* or *dataptr* is less than, respectively, the control or data part of the message, **maxlen** bytes are retrieved. In this case, the remainder of the message is left on the stream head read queue and a non-zero return value is provided, as described in **Errors**.

By default, **getmsg** processes the first available message on the stream head read queue. However, a user may choose to retrieve only high priority messages by setting the integer pointed by *flagsp* to **RS_HIPRI**. In this case, **getmsg** processes the next message only if it is a high priority message. If the integer pointed by *flagsp* is 0, **getmsg** retrieves any message available on the stream head read queue. In this case, on return, the integer pointed to by *flagsp* will be set to **RS_HIPRI** if a high priority message was retrieved, or 0 otherwise.

For **getpmsg**, the flags are different. *flagsp* points to a bitmask with the following mutually-exclusive flags defined: **MSG_HIPRI, MSG_BAND**, and **MSG_ANY**. Like **getmsg**, **getpmsg** processes the first available message on the stream head read queue. A user may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to **MSG_HIPRI** and the integer pointed to by *bandp* to 0. In this case, **getpmsg** will only process the next message if it is a high-priority message. In a similar manner, a user may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to **MSG_BAND** and the integer pointed to by *bandp* to the priority band of interest. In this case, **getpmsg** will only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a user just wants to get the first message off the queue, the integer pointed to by *flagsp* should be set to **MSG_ANY** and the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* will be set to **MSG_HIPRI** and the integer pointed to by *bandp* will be set to 0. Otherwise, the integer pointed to by *flagsp* will be set to **MSG_BAND** and the integer pointed to by *bandp* will be set to the priority band of the message.

If **O_NONBLOCK** is clear, **getmsg** blocks until a message of the type specified by *flagsp* is available on the stream head read queue. If **O_NONBLOCK** has been set and a message of the specified type is not present on the read queue, **getmsg** fails and sets **errno** to **EAGAIN**.

If a hangup occurs on the stream from which messages are to be retrieved, **getmsg** continues to operate normally, as described above, until the stream head read queue is empty. Thereafter, it returns 0 in the **len** fields of *ctlptr* and *dataptr*.

Return Values

On success, **getmsg** and **getpmsg** return a non-negative value:

0 indicates that a full message was read successfully.

MORECTL indicates that more control information is waiting for retrieval.

MOREDATA indicates that more data is waiting for retrieval.

(MORECTL | MOREDATA) indicates that both types of information remain.

Subsequent **getmsg** calls retrieve the remainder of the message. However, if a message of higher priority has come in on the stream head read queue, the next call to **getmsg** will retrieve that higher priority message before retrieving the remainder of the previously received partial message.

On failure, **getmsg** and **getpmsg** return –1 and set **errno** to identify the error.

Errors

In the following conditions, **getmsg** and **getpmsg** fail and set **errno** to:

EAGAIN The **O_NDELAY** flag is set, and no messages are available.

EBADF *fd* is not a valid file descriptor open for reading.

EBADMSG Queued message to be read is not valid for **getmsg**.

EFAULT *ctlptr*, *dataptr*, *bandp*, or *flagsp* points to a location outside the allocated address space.

EINTR A signal was caught during the **getmsg** system call.

EINVAL An illegal value was specified in *flagsp*, or the stream referenced by *fd* is linked under a multiplexor.

ENOSTR A stream is not associated with *fd*.

getmsg can also fail if a STREAMS error message had been received at the stream head before the call to **getmsg**. The error returned is the value contained in the STREAMS error message.

SEE ALSO

poll(BA_OS), **putmsg**(BA_OS), **read**(BA_OS), **write**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling. In this case, data input by one thread will not be available to others.

While one thread is blocked, siblings might still be executing.

NAME

getpid, **getpgrp**, **getppid**, **getpgid** – get process, process group, and parent process IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t pid);
```

DESCRIPTION

getpid returns the process **ID** of the calling process.

getpgrp returns the process group **ID** of the calling process.

getppid returns the parent process **ID** of the calling process.

getpgid returns the process group **ID** of the process whose process ID is equal to **pid**, or the process group **ID** of the calling process, if **pid** is equal to zero.

Return Values

On success, **getpgid** returns a process group **ID**. On failure, **getpgid** returns (**pid_t**) –1 and sets **errno** to identify the error.

Errors

In the following conditions, **getpgid** fails and sets **errno** to:

EPERM The process whose process **ID** is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group **ID** of that process from the calling process.

ESRCH There is no process with a process **ID** equal to *pid*.

NOTICES

Considerations for Threads Programming

These ID numbers are attributes of the containing process and are shared by sibling threads.

SEE ALSO

exec(BA_OS), **fork**(BA_OS), **getsid**(BA_OS), **signal**(BA_OS)

LEVEL

Level 1

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);

int setrlimit(int resource, const struct rlimit *rlp);
```

DESCRIPTION

Limits on the consumption of a variety of system resources by a process and each process it creates may be obtained with `getrlimit()` and set with `setrlimit()`.

Each call to either `getrlimit()` or `setrlimit()` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a user with appropriate privileges can raise a hard limit. Both hard and soft limits can be changed in a single call to `setrlimit()` subject to the constraints described above. Limits may have an infinite value of `RLIM_INFINITY`. *rlp* is a pointer to `struct rlimit` that includes the following members:

```
rlim_t    rlim_cur;    /* current (soft) limit */
rlim_t    rlim_max;    /* hard limit */
```

`rlim_t` is an arithmetic data type to which objects of type `int` and `off_t` can be cast without loss of value.

The possible resources, their descriptions, and the actions taken when current limit is exceeded, are summarized in the table below:

Resources	Description	Action
`RLIMIT_CORE`	The maximum size of a core file in bytes that may be created by a process. A limit of `0` will prevent the creation of a core file.	The writing of a core file will terminate at this size.
`RLIMIT_CPU`	The maximum amount of CPU time in seconds used by a process.	`SIGXCPU` is sent to the process. If the process is holding or ignoring `SIGXCPU`, the behavior is scheduling class defined.
`RLIMIT_DATA`	The maximum size of a process's heap in bytes.	The `malloc()` function will fail with errno set to `ENOMEM`.

Resources	Description	Action
RLIMIT_FSIZE	The maximum size of a file in bytes that may be created by a process. A limit of 0 will prevent the creation of a file.	SIGXFSZ is sent to the process. If the process is holding or ignoring SIGXFSZ, continued attempts to increase the size of a file beyond the limit will fail with errno set to EFBIG.
RLIMIT_NOFILE	The maximum number of open file descriptors that the process can have.	Functions that create new file descriptors will fail with errno set to EMFILE.
RLIMIT_STACK	The maximum size of a process's stack in bytes. The system will not automatically grow the stack beyond this limit.	SIGSEGV is sent to the process. If the process is holding or ignoring SIG-SEGV, or is catching SIG-SEGV and has not made arrangements to use an alternate stack [see sigaltstack(BA_OS)], the disposition of SIGSEGV will be set to SIG_DFL before it is sent.
†RLIMIT_AS	The maximum amount of a process's address space that is defined (in bytes).	The malloc() and mmap() functions will fail with errno set to ENOMEM. In addition, the automatic stack growth will fail with the effects outlined above.

Because limit information is stored in the per-process information, the shell builtin ulimit must directly execute this system call if it is to affect all future processes created by the shell.

The value of the current limit of the following resources affect these implementation defined constants:

Limit	Implementation Defined Constant
RLIMIT_FSIZE	FCHR_MAX
RLIMIT_NOFILE	OPEN_MAX

RETURN VALUE

Upon successful completion, the function getrlimit() returns a value of 0; otherwise, it returns a value of –1 and sets errno to indicate an error.

ERRORS

Under the following conditions, the functions getrlimit() and setrlimit() fail and set errno to:

EINVAL	if an invalid *resource* was specified; or in a `setrlimit()` call, the new `rlim_cur` exceeds the new `rlim_max`.
EPERM	if the limit specified to `setrlimit()` would have raised the maximum limit value, and the caller is not a user with appropriate privileges.

SEE ALSO

malloc(BA_OS), open(BA_OS), sigaltstack(BA_OS), signal(BA_ENV).

FUTURE DIRECTIONS

The resource `RLIMIT_AS` is marked level 2, and should be deprecated. It is not useful in all implementations since different implementations treat address space and size differently.

LEVEL

Level 1.

`RLIMIT_AS` is marked Level 2, effective September 30, 1993. It will be removed after the three year waiting period has expired.

NAME

getsid – get session ID

SYNOPSIS

```
#include <sys/types.h>

pid_t getsid(pid_t pid);
```

DESCRIPTION

The function `getsid()` returns the session ID of the process whose process ID is equal to *pid*. If *pid* is equal to `(pid_t)0`, `getsid()` returns the session ID of the calling process.

RETURN VALUE

Upon successful completion, the function `getsid()` returns the session ID of the specified process; otherwise, it returns a value of `(pid_t)-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `getsid()` fails and sets `errno` to:

`EPERM` if the process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the session ID of that process from the calling process.

`ESRCH` if there is no process with a process ID equal to *pid*.

SEE ALSO

exec(BA_OS), fork(BA_OS), getpid(BA_OS), getpgid(BA_OS), setpgid(BA_OS), setsid(BA_OS).

LEVEL

Level 1.

NAME

getuid, **geteuid**, **getgid**, **getegid** – get real user, effective user, real group, and effective group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid (void);

uid_t geteuid (void);

gid_t getgid (void);

gid_t getegid (void);
```

DESCRIPTION

getuid returns the real user ID of the calling process.

geteuid returns the effective user ID of the calling process.

getgid returns the real group ID of the calling process.

getegid returns the effective group ID of the calling process.

SEE ALSO

setuid(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

These ID numbers are attributes of the containing process and are shared by sibling threads.

NAME

ioctl – control device

SYNOPSIS

```
#include <sys/types.h>

int ioctl(int fildes, int request, ... /* arg */);
```

DESCRIPTION

The function **ioctl()** performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. *request* and an optional third argument (with varying type) are passed to the file designated by *fildes* and are interpreted by the device driver. This control is not frequently used on non-STREAMS devices, where the basic input/output functions are usually performed by the **read()** and **write()** functions.

For STREAMS files, specific functions are performed by the **ioctl** call as described in **streams(BA_DEV)**.

The argument *fildes* is an open file descriptor that refers to a device.

The argument *request* selects the control function to be performed and will depend on the device being addressed.

The argument *arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, there are generic functions that are provided by more than one device driver, for example, the general terminal interface [see **termio(BA_DEV)**].

When Mandatory Access Controls are running on the system, the invoking process must have MAC write access on *fildes* to do an **ioctl()**.

RETURN VALUE

Upon successful completion, the function **ioctl()** returns a value other than **–1** that depends upon the device control function; otherwise, a value of **–1** is returned and **errno** is set to indicate an error.

ERRORS

Under the following conditions, the function **ioctl()** fails and sets **errno** to:

EBADF if *fildes* is not a valid open file descriptor.

ENOTTY if *fildes* is not associated with a character-special file that accepts control functions.

EINTR if a signal was caught during the **ioctl()** operation.

The function **ioctl()** will also fail if the device driver detects an error. In this case, the error is passed through **ioctl()** without change to the caller. A particular device driver might not have all of the following error cases. Under the following conditions, requests to standard device drivers may fail and **errno** will be set to:

EINVAL if *request* or *arg* is not valid for this device.

EIO if some physical I/O error has occurred.

ENXIO if *request* and *arg* are valid for this device driver, but the service requested can not be performed on this particular sub-device.

SEE ALSO

termio(BA_DEV), termios(BA_OS), streams(BA_DEV).

See also the specific device reference documents and generic devices such as the general terminal interface.

LEVEL

Level 1.

NAME

kill – send a signal to a process or a group of processes

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int sig);
```

DESCRIPTION

kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in **signal** [see **signal**(BA_OS)], or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

In order to send the signal to the target process (*pid*), the sending process must have permission to do so, subject to the following ownership restrictions:

> The real or effective user ID of the sending process must match the real or saved [from **exec**] user ID of the receiving process, unless the sending process has the **P_OWNER** privilege, or *sig* is **SIGCONT** and the sending process has the same session ID as the receiving process.

The process with **ID** 0 and the process with **ID** 1 are special processes and will be referred to below as proc0 and proc1, respectively.

If *pid* is greater than 0, *sig* will be sent to the process whose process *ID* is equal to *pid*, subject to the ownership restrictions, above. *pid* may equal 1.

If *pid* is negative but not **(pid_t)-1**, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid* and for which the process has permission to send a signal.

If *pid* is **0**, *sig* will be sent to all processes excluding proc0 and proc1 whose process group ID is equal to the process group ID of the sender. Permission is needed to send a signal to process groups.

If *pid* is **(pid_t)-1** and the sending process does not have the **P_OWNER** privilege, *sig* will be sent to all processes excluding proc0 and proc1 whose real user ID is equal to the effective user ID of the sender.

If *pid* is **(pid_t)-1** and the sending process has the **P_OWNER** privilege, *sig* will be sent to all processes excluding proc0 and proc1.

Return Values

On success, **kill** returns 0. On failure, **kill** returns –1, sets **errno** to identify the error, and sends no signal.

Errors

In the following conditions, **kill** fails and sets **errno** to:

EINVAL	*sig* is not a valid signal number.
EPERM	*sig* is **SIGKILL** and *pid* is **(pid_t)1** (i.e., *pid* specifies proc1).

EPERM The sending process does not have the **P_OWNER** privilege, the real or effective user ID of the sending process does not match the real or saved user ID of the receiving process, and the calling process is not sending **SIGCONT** to a process that shares the same session ID.

ESRCH No process or process group can be found corresponding to that specified by *pid*.

SEE ALSO

getsid(BA_OS), **sigaction**(BA_OS), **signal**(BA_OS) **sigsend**(BA_OS)

LEVEL

Level 1.

NOTICES

sigsend is a more versatile way to send signals to processes. The user is encouraged to use **sigsend** instead of **kill**.

NAME

link – link to a file

SYNOPSIS

```
#include <unistd.h>

int link(const char *path1, const char *path2);
```

DESCRIPTION

The function `link()` atomically creates a new link (directory entry) for the existing file.

The *path1* argument points to a pathname naming an existing file. The *path2* argument points to a pathname naming the new directory entry to be created. The `link()` function will atomically create a new link for the existing file and the link count of the file is incremented by one.

If *path1* names a directory, `link()` will fail unless the process has appropriate privileges and the implementation supports making links to directories.

Upon successful completion, the function `link()` marks for update the `st_ctime` field of the file. Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.

RETURN VALUE

Upon successful completion, the function `link()` returns a value of `0`; otherwise, it returns a value of `-1`, no link is created, and the link count of the file will remain unchanged after the call. The function sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `link()` fails and sets `errno` to:

`ENOTDIR`	if a component of either path prefix is not a directory.
`ENOENT`	if a component of either pathname should exist but does not, or the file named by *path1* does not exist or *path1* or *path2* points to an empty string.
`EACCES`	if a component of either path prefix denies search permission, or if the requested link requires writing in a directory with a mode that denies write permission.
`EEXIST`	if the link named by *path2* exists.
`ELOOP`	if too many symbolic links are encountered while translating either path.
`EPERM`	if the file named by *path1* is a directory and the process does not have appropriate privileges.
`EXDEV`	if the link named by *path2* and the file named by *path1* are on different logical devices (file systems) and the implementation does not permit cross-device links, or if *path* refers to a named stream.
`EROFS`	if the requested link requires writing in a directory on a read-only file system.

`EMLINK` if the number of links after execution would exceed `{LINK_MAX}`, the maximum number of links to a single file.

`ENOSPC` if the directory that would contain the link cannot be extended.

`ENAMETOOLONG`
if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

SEE ALSO

rename(BA_OS), symlink(BA_OS), unlink(BA_OS).

LEVEL

Level 1.

NAME

lockf – record locking on files

SYNOPSIS

```
#include <unistd.h>

int lockf (int fildes, int function, long size);
```

DESCRIPTION

lockf locks sections of a file. Advisory or mandatory write locks depend on the mode bits of the file; see **chmod**(BA_OS). Other processes that try to lock the locked file section either get an error or go to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. See **fcntl** for more information about record locking.

fildes is an open file descriptor. The file descriptor must have **O_WRONLY** or **O_RDWR** permission to establish locks with this function call.

function is a control value that specifies the action to be taken. The permissible values for *function* are defined in **unistd.h** as follows:

```
#define F_ULOCK  0  /* unlock previously locked section */
#define F_LOCK   1  /* lock section for exclusive use */
#define F_TLOCK  2  /* test & lock section for exclusive use */
#define F_TEST   3  /* test section for other locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. **F_LOCK** and **F_TLOCK** both lock a section of a file if the section is available. **F_ULOCK** removes locks from a section of the file.

size is the number of contiguous bytes to be locked or unlocked. The resource to be locked or unlocked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file to be locked as such locks may exist past the end-of-file.

The sections locked with **F_LOCK** or **F_TLOCK** may, in whole or in part, contain or be contained by a previously locked section for the same process. Locked sections will be unlocked starting at the the point of the offset through *size* bytes or to the end of file if *size* is (**off_t**) 0. When this occurs, or if this occurs in adjacent sections, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and **F_TLOCK** requests differ only by the action taken if the resource is not available. **F_LOCK** will cause the calling process to sleep until the resource is available. **F_TLOCK** will cause the function to return a –1 and set **errno** to **EACCES** if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an **errno** is set to **EDEADLK** and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by requesting another process's locked resource. Thus calls to **lockf** or **fcntl** scan for a deadlock before sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The **alarm** system call may be used to provide a timeout facility in applications that require this facility.

Return Values

On success, **lockf** returns 0. On failure, **lockf** returns –1 and sets **errno** to indicate the error.

Errors

lockf will fail if one or more of the following are true:

EBADF	*fildes* is not a valid open descriptor.
EAGAIN	*cmd* is **F_TLOCK** or **F_TEST** and the section is already locked by another process.
EDEADLK	*cmd* is **F_LOCK** and a deadlock would occur.
EDEADLK	*cmd* is **F_LOCK**, **F_TLOCK**, or **F_ULOCK** and the number of entries in the lock table would exceed the number allocated on the system.
EACCES	If *function* is **F_TLOCK** or **F_TEST** and the section is already locked by another process.

SEE ALSO

chmod (BA_OS), **close** (BA_OS), **creat** (BA_OS), **fcntl** (BA_OS), **open** (BA_OS), **read** (BA_OS), **write** (BA_OS)

LEVEL

Level 1

NOTICES

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data that is/was locked. The standard I/O package is the most common source of unexpected buffering.

Because in the future the variable **errno** will be set to **EAGAIN** rather than **EACCES** when a section of a file is already locked by another process, portable application programs should expect and test for either value.

NAME

lseek – move read / write file pointer

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (int fildes, off_t offset, int whence);
```

DESCRIPTION

lseek moves a read / write file pointer. *fildes* is a file descriptor returned from a **creat**, **open**, **dup**, **fcntl**, **pipe**, or **ioctl** system call. **lseek** sets the file pointer associated with *fildes* as follows:

If *whence* is **SEEK_SET**, the pointer is set to *offset* bytes.

If *whence* is **SEEK_CUR**, the pointer is set to its current location plus *offset*.

If *whence* is **SEEK_END**, the pointer is set to the size of the file plus *offset*.

On success, **lseek** returns the resulting pointer location, as measured in bytes from the beginning of the file.

lseek allows the file pointer to be set beyond the existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data return bytes of value 0 until data is written into the gap.

Return Values

On success, **lseek** returns a non-negative integer indicating the file pointer value. On failure, **lseek** returns –1, sets **errno** to identify the error, and the file pointer remains unchanged.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

Errors

In the following conditions, **lseek** fails and sets **errno** to:

EBADF	*fildes* is not an open file descriptor.
ESPIPE	*fildes* is associated with a pipe or fifo.
EINVAL	The resulting file pointer would be negative. *fildes* is a remote file descriptor accessed using NFS, the Network File System, and the resulting file pointer would be negative.
ENOSYS	The device for **fstype** does not support **lseek**.

USAGE

Normally, applications should use the stdio routines to open, close, read, write, and manipulate files. Therefore, an application using the **fopen** stdio routine to open a file would use the **fseek** stdio routine rather than the function **lseek**. The function **lseek** allows the file pointer to be set beyond the existing data in the file. If data are later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes of value **0** until data are written into the gap.

SEE ALSO

creat (BA_OS), **fcntl** (BA_OS), **open** (BA_OS)

LEVEL

Level 1.

NOTICES

On systems that support Remote File Sharing (RFS), the behavior of **lseek** is different for files accessed using RFS. For other files, the file pointer can be positioned to negative values where attempts to **write** will fail. For FIFOs, **lseek** returns successfully, for both positive and negative offsets, instead of failing with **ESPIPE**. These semantics can be used to identify files that are being accessed using RFS.

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling. For example, the position of the file pointer is maintained per file descriptor, not per thread.

NAME

malloc, **free**, **realloc**, **calloc**, – memory allocator

SYNOPSIS

```
#include <stdlib.h>
void *malloc (size_t size);
void free (void *ptr);
void *realloc (void *ptr, size_t size);
void *calloc (size_t nelem, size_t elsize);
‡int mallopt(int cmd, int value);
‡struct mallinfo mallinfo(void);
```

DESCRIPTION

malloc and **free** provide a simple general-purpose memory allocation package. **malloc** returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to **free** is a pointer to a block previously allocated by **malloc**, **calloc** or **realloc**. After **free** is performed, this space is made available for further allocation. If *ptr* is **NULL**, no action occurs.

Undefined results will occur if the space assigned by **malloc** is overrun or if some random pointer is handed to **free**.

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If *ptr* is **NULL**, **realloc** behaves like **malloc** for the specified size. If *size* is zero and *ptr* is not a null pointer, the object pointed to is freed.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

The functions **mallopt** and **mallinfo** are marked Level 2 in this issue of SVID. The use of these functions should be discouraged.

The function **mallopt** plus the function **mallinfo** allow tuning the allocation algorithm at execution time.

The function **mallopt** initiates a mechanism that can be used to allocate small blocks of memory quickly. Using this scheme, a large-group (called a *holding-block*) of these small-blocks is allocated at one time. Then, each time a program requests a small amount of memory from **malloc**, a pointer to one of the *pre-allocated* small-blocks is returned. Different holding-blocks are created for different sizes of small-blocks and are created when needed.

The function **mallopt** allows the programmer to set three parameters to maximize efficient small-block allocation for a particular application.

The function **mallopt** may be called repeatedly, but the parameters may not be changed after the first small-block is allocated from a holding-block. If **mallopt** is called again after the first small-block is allocated using the small-block algorithm, it will return an error.

The function **mallinfo** can be used during program development to determine the best settings of these parameters for a particular application. The function **mallinfo** should not be called until after some storage has been allocated using **malloc**. The function **mallinfo** provides information describing space usage. It returns a **mallinfo** structure.

Errors

If there is no available memory, **malloc**, **realloc**, and **calloc** return a null pointer. When **realloc** returns **NULL**, the block pointed to by *ptr* is left intact. If *size*, *nelem*, or *elsize* is 0, a unique pointer to the arena is returned. If **mallopt** is called after any allocation from a holding-block or if the arguments *cmd* or *value* are invalid, **mallopt** returns a non-zero value; otherwise, it returns a value of **0**.

USAGE

You can control whether the contents of the freed space are destroyed or left undisturbed [see **mallopt**].

FUTURE DIRECTIONS

The functions **mallopt** and **mallinfo** are marked Level 2 effective September 30, 1993. The use of these functions is deprecated; they will be removed from the next issue of SVID.

LEVEL

Level 1.

The functions **mallopt** and **mallinfo** are marked Level 2 effective, September 30, 1993.

NAME

mkdir – make a directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

DESCRIPTION

mkdir creates a new directory named by the pathname pointed to by *path*. The mode of the new directory is initialized from *mode* [see **chmod**(BA_OS) for the values of mode.]

The protection part of the *mode* argument is modified by the process's file create mask.

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID, or if the **S_ISGID** bit is set in the parent directory, then the group ID of the directory is inherited from the parent. The **S_ISGID** bit of the new directory is inherited from the parent directory.

If *path* is a symbolic link, it is not followed.

The newly created directory is empty with the exception of entries for itself (**.**) and its parent directory (**..**).

Return Values

On success, **mkdir** returns 0 and marks for update the **st_atime**, **st_ctime** and **st_mtime** fields of the directory. Also, the **st_ctime** and **st_mtime** fields of the directory that contains the new entry are marked for update.

On failure, **mkdir** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **mkdir** fails and sets **errno** to:

EACCES Search permission is denied on a component of the path prefix.

EACCES Write permission is denied on the parent directory in which the directory is to be created.

EEXIST The named file already exists.

EIO An I/O error has occurred while accessing the file system.

ELOOP Too many symbolic links were encountered in translating *path*.

EMLINK The maximum number of links to the parent directory would be exceeded.

ENAMETOOLONG
The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.

ENOENT A component of the path prefix does not exist or is a null pathname.

ENOSPC	No free space is available on the device containing the directory.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The path prefix resides on a read-only file system.

SEE ALSO

chmod(BA_OS), **directory**(BA_OS) **rmdir**(BA_OS) **umask**(BA_OS).

LEVEL

Level 1.

NAME

mkfifo – create a new FIFO

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

DESCRIPTION

The `mkfifo()` routine creates a new FIFO special file named by the pathname pointed to by *path*. The mode of the new FIFO is initialized from *mode*. The file permission bits of the *mode* argument are modified by the process's file creation mask.

The FIFO's owner ID is set to the process's effective user ID. The FIFO's group ID is set to the process's effective group ID unless the set-group-ID flag of the FIFO's parent directory is set; in that case it is initialised to the group ID of the parent directory.

Bits other than the file permission bits in *mode* are ignored.

Upon successful completion, the function `mkfifo()` marks for update the `st_atime`, `st_ctime` and `st_mtime` field of the file. Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.

RETURN VALUE

Upon successful completion, a value of zero is returned; otherwise, a value of `-1` is returned and `errno` is set to indicate an error.

ERRORS

`EACESS`	A component of the path prefix denies search permission, or write permission is denied on the parent directory.
`EEXIST`	The named file already exists.
`EIO`	An I/O error occurred while accessing the file system.
`ELOOP`	if too many symbolic links are encountered in translating *path*.
`ENOENT`	A component of the path prefix does not exist, or *path* points to an empty string.
`ENOSPC`	if the directory that would contain the FIFO cannot be extended or the file system is out of file allocation resources.
`ENOTDIR`	A component of the *path* prefix is not a directory.
`EROFS`	The directory in which the file is to be created is located on a read-only file system.
`ENAMETOOLONG`	if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

SEE ALSO

chmod(BA_OS), exec(BA_OS), mkdir(BA_OS), mknod(BA_OS), umask(BA_OS)

LEVEL

Level 1.

NAME

mknod – make a directory, or a special or ordinary file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);
```

DESCRIPTION

mknod creates a new file named by the path name pointed to by *path*. The file type and permissions of the new file are initialized from *mode*.

The file type is specified in *mode* by the **S_IFMT** bits, which must be set to one of the following values:

S_IFIFO	fifo special
S_IFCHR	character special
S_IFDIR	directory
S_IFBLK	block special
S_IFREG	ordinary file

The file access permissions are specified in *mode* by the 0007777 bits, and may be constructed by an OR of the following values:

S_ISUID	Set user ID on execution.
S_ISGID	Set group ID on execution if # is **7**, **5**, **3**, or **1** Enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**
S_ISVTX	Save text image after execution.
S_IRWXU	Read, write, execute by owner.
S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IXUSR	Execute (search if a directory) by owner.
S_IRWXG	Read, write, execute by group.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IXGRP	Execute by group.
S_IRWXO	Read, write, execute (search) by others.
S_IROTH	Read by others.
S_IWOTH	Write by others
S_IXOTH	Execute by others.

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process. However, if the **S_ISGID** bit is set in the parent directory, then the group ID of the file is inherited from the parent. If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the **S_ISGID** bit is cleared.

The access permission bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared [see **umask**(BA_OS)]. If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

mknod checks to see if the driver has been installed and whether or not it is an old-style driver. If the driver is installed and it is an old-style driver, the minor number is limited to 255. If it's not an old-style driver, then it must be a new-style driver or uninstalled, and the minor number is limited to the current value of the **MAXMINOR** tunable. Of course, this tunable is set to 255 by default. If the range check fails, **mknod** fails with **EINVAL**.

mknod may be invoked only by a privileged user for file types other than FIFO special.

If *path* is a symbolic link, it is not followed.

Return Values

If **mknod** succeeds, it returns 0. If **mknod** fails, it returns –1 and sets **errno** to identify the error.

Errors

mknod fails and creates no new file if one or more of the following are true:

EEXIST The named file exists.

EINVAL *dev* is invalid.

EFAULT *path* points outside the allocated address space of the process.

ELOOP Too many symbolic links were encountered in translating *path*.

EMULTIHOP Components of *path* require hopping to multiple remote machines and the file system type does not allow it.

ENAMETOOLONG
The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.

ENOTDIR A component of the path prefix is not a directory.

ENOENT A component of the path prefix does not exist or is a null pathname.

EPERM The effective user ID of the process is not super-user.

EROFS The directory in which the file is to be created is located on a read-only file system.

ENOSPC No space is available.

EINTR A signal was caught during the **mknod** system call.

ENOLINK *path* points to a remote machine and the link to that machine is no longer active.

SEE ALSO

chmod(BA_OS), **exec**(BA_OS), **mkdir**(BU_CMD), **stat**(BA_OS), **umask**(BA_OS)

LEVEL

Level 1.

NAME

mount – mount a file system

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mount.h>

int mount (const char *spec, const char *dir, int mflag,
```

DESCRIPTION

mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to path names. *fstyp* is the file system type number. If both the **MS_DATA** and **MS_FSS** flag bits of *mflag* are off, the file system type defaults to the root file system type. Only if either flag is on is *fstyp* used to indicate the file system type.

If the **MS_DATA** flag is set in *mflag* the system expects the *dataptr* and *datalen* arguments to be present. Together they describe a block of file-system specific data at address *dataptr* of length *datalen*. This is interpreted by file-system specific code within the operating system and its format depends on the file system type. If a particular file system type does not require this data, *dataptr* and *datalen* should both be zero. Note that **MS_FSS** is obsolete and is ignored if **MS_DATA** is also set, but if **MS_FSS** is set and **MS_DATA** is not, *dataptr* and *datalen* are both assumed to be zero.

After a successful call to **mount**, all references to the file *dir* refer to the root directory on the mounted file system.

The low-order bit of *mflag* is used to control write permission on the mounted file system: if **1**, writing is forbidden; otherwise writing is permitted according to individual file accessibility.

mount may be invoked only by a process with the **P_MOUNT** privilege. It is intended for use only by the **mount** utility.

Return Values

On success, **mount** returns 0. On failure, **mount** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **mount** fails and sets **errno** to:

EPERM	The calling process does not have the appropriate privilege.
EBUSY	*dir* is currently mounted on, is someone's current working directory, or is otherwise busy.
EBUSY	The device associated with *spec* is currently mounted.
EBUSY	There are no more mount table entries.
EINVAL	The super block has an invalid magic number or the *fstyp* is invalid.
ELOOP	Too many symbolic links were encountered in translating *spec* or *dir*.

ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.
ENOENT	None of the named files exists or is a null pathname.
ENOTDIR	A component of a path prefix is not a directory.
ENOTBLK	*spec* is not a block special device.
ENXIO	The device associated with *spec* does not exist.
ENOTDIR	*dir* is not a directory.
EROFS	*spec* is write protected and *mflag* requests write permission.
ENOSPC	The file system state in the super-block is not **FsOKAY** and there is no space left on the device.

USAGE

mount is not recommended for use by application programs.

SEE ALSO

mount(AS_CMD), **umount**(BA_OS)

LEVEL

Level 1.

NAME

open – open for reading or writing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *path, int oflag, . . . /* mode_t mode */);
```

DESCRIPTION

open opens a file descriptor for the file named *path* and sets the file status flags according to the value of *oflag*. *oflag* values are constructed by OR-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NONBLOCK This flag may affect subsequent reads and writes [see **read**(BA_OS) and **write**(BA_OS)].

When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

If **O_NONBLOCK** is set: An **open** for reading-only will return without delay; an **open** for writing-only will return an error if no process currently has the file open for reading.

If **O_NONBLOCK** is clear: An **open** for reading-only will block until a process opens the file for writing; an **open** for writing-only will block until a process opens the file for reading.

When opening a file associated with a terminal line:

If **O_NONBLOCK** is set: The open will return without waiting for the device to be ready or available; subsequent behavior of the device is device specific.

If **O_NONBLOCK** is clear: The open will block until the device is ready or available.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each **write** will wait for both the file data and file status to be

the group ID of the new file does not match the effective group ID or one of the supplementary groups IDs, the **S_ISGID** bit is cleared. The access permission bits of the file mode are set to the value of *mode*, modified as follows [see **creat**(BA_OS)]:

All bits set in the file mode creation mask of the process are cleared [see **umask**(BA_OS)].

The ''save text image after execution bit'' of the mode is cleared [see **chmod**(BA_OS)].

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged. **O_TRUNC** has no effect on special files or directories.

O_EXCL If **O_EXCL** and **O_CREAT** are set, **open** will fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other processes executing **open** naming the same filename in the same directory with **O_EXCL** and **O_CREAT** set.

When opening a STREAMS file, *oflag* may be constructed from **O_NONBLOCK** OR-ed with either **O_RDONLY**, **O_WRONLY ,** or **O_RDWR**. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of **O_NONBLOCK** affects the operation of STREAMS drivers and certain system calls [see **read**(BA_OS), **getmsg**(BA_OS), **putmsg**(BA_OS), and **write**(BA_OS)]. For drivers, the implementation of **O_NONBLOCK** is device specific. Each STREAMS device driver may treat these options differently.

When **open** is invoked to open a named stream, and the **connld** module [see **connld**] has been pushed on the pipe, **open** blocks until the server process has issued an **I_RECVFD ioctl** [see **streams**(BA_DEV)] to receive the file descriptor.

If *path* is a symbolic link and **O_CREAT** and **O_EXCL** are set, the link is not followed.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is the lowest numbered file descriptor available and is set to remain open across **exec** system calls [see **fcntl**(BA_OS)].

Certain flag values can be set following **open** as described in **fcntl**.

Using **open** on a file adds a reference to the file. This guarantees that the file will continue to be visible to the process until it closes it, even if the file is removed from the directory by **unlink**.

Return Values

On success, **open** returns the file descriptor of the open file and:

If **O_CREAT** is set and the file did not previously exist, **open** marks for update the **st_atime**, **st_ctime** and **st_mtime** fields of the file and the **st_ctime** and **st_mtime** fields of the parent directory.

If **O_TRUNC** is set and the file did previously exist, **open** marks for update the **st_ctime** and **st_mtime** fields of the file.

On failure, **open** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **open** fails and sets **errno** to:

EACCES The file does not exist and write permission is denied by the parent directory of the file to be created.

EACCES **O_CREAT** or **O_TRUNC** is specified and write permission is denied.

EACCES A component of the path prefix denies search permission.

EACCES *oflag* permission is denied for an existing file.

EAGAIN The file exists, mandatory file / record locking is set, and there are outstanding record locks on the file [see **chmod**(BA_OS)].

EEXIST **O_CREAT** and **O_EXCL** are set, and the named file exists.

EINTR A signal was caught during the **open** system call.

EIO A hangup or error occurred during the open of the STREAMS-based device.

EISDIR The named file is a directory and *oflag* is write or read / write.

ELOOP Too many symbolic links were encountered in translating *path*.

EMFILE The process has too many open files

ENAMETOOLONG
The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.

ENFILE The system file table is full.

ENOENT **O_CREAT** is not set and the named file does not exist.

ENOENT **O_CREAT** is set and a component of the path prefix does not exist or is the null pathname.

ENOSPC **O_CREAT** and **O_EXCL** are set, and the file system is out of inodes.

ENOSPC **O_CREAT** is set and the directory that would contain the file cannot be extended.

ENOSR Unable to allocate a stream.

ENOTDIR A component of the path prefix is not a directory.

ENXIO The named file is a character special or block special file, and the device associated with this special file does not exist.

ENXIO **O_NONBLOCK** is set, the named file is a FIFO, **O_WRONLY** is set, and no process has the file open for reading.

ENXIO A STREAMS module or driver open routine failed.

EROFS The named file resides on a read-only file system and either **O_WRONLY**, **O_RDWR**, **O_CREAT**, or **O_TRUNC** is set in *oflag* (if the file does not exist).

ETXTBSY The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read / write.

USAGE

The **O_EXCL** flag is only a modifier to the **O_CREAT** flag and has no other meaning. The concept of **exclusive open** is not supported by the operating system. Cooperating processes can coordinate their access to a file by file and record locking or by other mechanisms.

SEE ALSO

chmod(BA_OS), **close**(BA_OS), **creat**(BA_OS), **fcntl**(BA_OS), **fopen**(BA_OS), **lseek**(BA_OS), **read**(BA_OS), **streams**(BA_DEV), **umask**(BA_OS), **write**(BA_OS).

LEVEL

Level 1.

The enforcement mode of file and record locking has moved to Level 2 effective September 30, 1989.

NOTICES

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

While one thread is blocked, siblings might still be executing.

Access rights are an attribute of the containing process and are shared by sibling threads.

NAME

pause – suspend process until signal

SYNOPSIS

```
#include <unistd.h>

int pause(void);
```

DESCRIPTION

pause suspends the calling process until it receives a signal of any type. The signal must be one that is not currently set to be ignored.

If the signal causes termination of the process, **pause** does not return.

Return Values

If the signal is caught by the calling process and control is returned from the signal-catching function [see **signal**(BA_OS)], the calling process resumes execution from the point of suspension with a return value of –1 from **pause** and **errno** set to **EINTR**.

Errors

In the following conditions, the calling process resumes from the point of suspension with **errno** set to:

EINTR A signal was caught by the calling process.

SEE ALSO

alarm(BA_OS), **kill**(BA_OS), **signal**(BA_OS), **wait**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

While one thread is blocked, siblings might still be executing. See **signal**(BA_OS) for further details of signal delivery.

NAME

pipe – create an interprocess channel

SYNOPSIS

```
#include <unistd.h>

int pipe(int fildes[2]);
```

DESCRIPTION

pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. The files associated with *fildes*[0] and *fildes*[1] are streams and are both opened for reading and writing. The **O_NONBLOCK** flag is cleared.

A read from *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis and a read from *fildes*[1] accesses the data written to *fildes*[0] also on a FIFO basis.

The **FD_CLOEXEC** flag will be clear on both file descriptors.

If **pipe** succeeds, it marks for update the **st_atime**, **st_ctime**, and **st_mtime** fields of the pipe.

Return Values

On success, **pipe** returns 0. On failure, **pipe** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **pipe** fails and sets **errno** to:

EMFILE The maximum number of file descriptors are currently open.

ENFILE A file table entry could not be allocated.

SEE ALSO

fcntl(BA_OS), **read**(BA_OS), **streams**(BA_DEV), **write**(BA_OS)

LEVEL

Level 1.

NOTICES

Since a pipe is bi-directional, there are two separate flows of data. Therefore, the size (**st_size**) returned by a call to **fstat** with argument *fildes*[0] or *fildes*[1] is the number of bytes available for reading from *fildes*[0] or *fildes*[1] respectively. Previously, the size (**st_size**) returned by a call to **fstat** with argument *fildes*[1] (the write-end) was the number of bytes available for reading from *fildes*[0] (the read-end). See **stat**(2).

NAME

poll – input/output multiplexing

SYNOPSIS

```
#include <poll.h>
int poll(struct pollfd fds[], unsigned long nfds, int timeout);
```

DESCRIPTION

`poll()` provides users with a mechanism for multiplexing input/output over a set of file descriptors. `poll()` identifies those file descriptors on which a user can read or write data, or on which certain events have occurred. A user can read data using `read()` [see read(BA_OS)] and write data using `write()` [see write(BA_OS)]. For STREAMS file descriptors, a user can also receive messages using `getmsg()` and `getpmsg()` [see getmsg(BA_OS) and `getpmsg()` in getmsg(BA_OS)] and send messages using `putmsg()` and `putpmsg()` [see putmsg(BA_OS) and `putpmsg()` in putmsg(BA_OS)].

fds specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are `pollfd` structures which contain the following members:

```
int fd;             /* file descriptor */
short events;       /* requested events */
short revents;      /* returned events */
```

where `fd` specifies an open file descriptor and `events` and `revents` are bitmasks constructed by OR-ing a combination of the following event flags:

`POLLIN`

POLLRDBAND, or POLLPRI are not mutually exclusive. This flag is only valid in the revents bitmask; it is not used in the events field.

POLLNVAL The specified fd value is invalid. This flag is only valid in the revents field; it is not used in the events field.

For each element of the array pointed to by *fds*, poll() examines the given file descriptor for the event(s) specified in events. The number of file descriptors to be examined is specified by *nfds*.

If the value of fd is less than zero, events is ignored and revents is set to zero in that entry on return from poll().

The results of the poll() query are stored in the revents field in the pollfd structure. Bits are set in the revents bitmask to indicate which of the requested events are true. If none of the requested events are true, none of the specified bits is set in revents when the poll() call returns. The event flags POLLHUP, POLLERR, and POLLNVAL are always set in revents if the conditions they indicate are true; this occurs even though these flags were not present in events.

If none of the defined events have occurred on any selected file descriptor, poll() waits at least *timeout* milliseconds for an event to occur on any of the selected file descriptors. On a computer where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. If the value of *timeout* is 0, poll() returns immediately. If the value of *timeout* is -1, poll() blocks until a requested event occurs or until the call is interrupted. poll() is not affected by the O_NDELAY and O_NONBLOCK flags.

RETURN VALUE

Upon successful completion, the function poll() returns a non-negative value. A positive value indicates the total number of file descriptors that have been selected (*i.e.*, file descriptors for which the revents field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, the function poll() returns a value of -1 and sets errno to indicate an error.

ERRORS

Under the following conditions, the function poll() fails and sets errno to:

EAGAIN if the allocation of internal data structures failed but request should be attempted again.

EINTR if a signal was caught during the poll() system call.

EINVAL if the argument *nfds* is less than zero or greater than {OPEN_MAX}.

SEE ALSO

getmsg(BA_OS), putmsg(BA_OS), read(BA_OS), streams(BA_DEV), write(BA_OS).

LEVEL

Level 1.

NAME

popen, pclose – initiate pipe to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

int pclose(FILE *strm);
```

DESCRIPTION

The function `popen()` creates a pipe between the calling program and the command to be executed.

The arguments to `popen()` are pointers to null-terminated strings containing, respectively, a command line [see system(BA_OS)] and an I/O mode, either `"r"` for reading or `"w"` for writing.

The function `popen()` returns a stdio-stream pointer such that one can write to the standard input of the command if the I/O mode is `"w"` by writing to the file *strm*; and one can read from the standard output of the command if the I/O mode is `"r"` by reading from the file *strm*. If *command* cannot be executed, the read or write will fail.

A stdio-stream opened by the function `popen()` should be closed by the function `pclose()`, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type `"r"` command may be used as an input filter and a type `"w"` command as an output filter.

RETURN VALUE

If files or processes cannot be created the function `popen()` returns `NULL`.

If *strm* is not associated with a `popen()` command, the function `pclose()` returns a value of `-1`.

ERRORS

Under the following conditions, the function `pclose()` fails and sets `errno` to:

`ECHILD` if the status of the child process could not be obtained.

USAGE

The `fseek()` routine should not be used with a stdio-stream opened by the function `popen()`.

SEE ALSO

fclose(BA_OS), fopen(BA_OS), fseek(BA_OS), pipe(BA_OS), system(BA_OS), wait(BA_OS).

LEVEL

Level 1.

NAME

pread – atomic position and read

SYNOPSIS

```
int pread(int fd, char *buf, int nbytes, off_t offset);
```

DESCRIPTION

The **pread** system call does an atomic position-and-read, eliminating the necessity of using a locking mechanism when both operations are desired and file descriptors are shared. **pread** is analogous to **read** but takes a fourth argument, *offset*. The read is done as if an **lseek** to *offset* (from the beginning of the file) were done first. Note that (though the semantics are analogous) an **lseek** is not actually performed; the file pointer is not affected by **pread**. The read of *nbytes* then starts at the specified offset.

The atomicity of **pread** enables processes or threads that share file descriptors to read from a shared file at a particular offset without using a locking mechanism that would be necessary to achieve the same result in separate **lseek** and **read** system calls. Atomicity is required as the file pointer is shared and one thread might move the pointer using *lseek* after another process completes an **lseek** but prior to the **read**.

Return Values

Upon successful completion, **pread** returns the number of bytes actually read and placed in *buf*. A value of 0 is returned when an end-of-file has been reached. Otherwise a -1 and an error is returned.

Errors

In the following conditions, **pread** fails and set **errno** to:

EACCES	*fildes* is open to a dynamic device and read permission is denied.
EAGAIN	Mandatory file/record locking was set, **O_NDELAY** or **O_NONBLOCK** was set, and there was a blocking record lock.
EAGAIN	Total amount of system memory available when reading via raw I/O is temporarily insufficient.
EAGAIN	No data is waiting to be read on a file associated with a tty device and **O_NONBLOCK** was set.
EAGAIN	No message is waiting to be read on a stream and **O_NDELAY** or **O_NONBLOCK** was set.
EBADF	*fildes* is not a valid file descriptor open for reading.
EBADMSG	Message waiting to be read on a stream is not a data message.
EDEADLK	The **pread** was going to go to sleep and cause a deadlock to occur.
EFAULT	*buf* points outside the allocated address space.
EINTR	A signal was caught during the **pread** system call.
EINVAL	Attempted to read from a stream linked to a multiplexor.

EINVAL	The resulting file pointer would be negative.
EINVAL	*fildes* is a remote file descriptor accessed using NFS, the Network File System, and the resulting file pointer would be negative.
EIO	A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.
EIO	*fildes* is open to a device that is in the process of closing.
ENOLCK	The system record lock table was full, so the **pread** could not go to sleep until the blocking record lock was removed.
ENOLINK	*fildes* is on a remote machine and the link to that machine is no longer active.
ESPIPE	*fildes* is associated with a pipe or fifo.
ENOSYS	The device for *fstype* does not support seek operations.

SEE ALSO

lseek(BA_OS), **pwrite**(BA_OS), **read**(BA_OS)

LEVEL

Level 1

NOTICES

pread updates the time of last access [see **stat**(BA_OS)] of the file.

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

While one thread is blocked, siblings might still be executing.

NAME

putmsg, putpmsg – send a message on a stream

SYNOPSIS

```
#include <stropts.h>

int putmsg(int fd, const struct strbuf *ctlptr,
      const struct strbuf *dataptr, int flags);

int putpmsg(int fd, const struct strbuf *ctlptr,
      const struct strbuf *dataptr, int band, int flags);
```

DESCRIPTION

putmsg creates a message from user-specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

The function **putpmsg** does the same thing as **putmsg**, but provides the user the ability to send messages in different priority bands. Except where noted, all information pertaining to **putmsg** also pertains to **putpmsg**.

fd specifies a file descriptor referencing an open stream. *ctlptr* and *dataptr* each point to a **strbuf** structure, which contains the following members:

```
int maxlen;     /* not used */
int len;        /* length of data */
void *buf;      /* ptr to buffer */
```

ctlptr points to the structure describing the control part, if any, to be included in the message. The **buf** field in the **strbuf** structure points to the buffer where the control information resides, and the **len** field indicates the number of bytes to be sent. The **maxlen** field is not used in **putmsg** [see **getmsg**(BA_OS)]. In a similar manner, *dataptr* specifies the data, if any, to be included in the message. *flags* indicates what

and sets **errno** to **EINVAL**. If flags is set to **MSG_BAND**, then a message is sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to **MSG_BAND**, no message is sent and 0 is returned.

Normally, **putmsg** will block if the stream write queue is full due to internal flow control conditions. For high-priority messages, **putmsg** does not block on this condition. For other messages, **putmsg** does not block when the write queue is full and **O_NONBLOCK** is set. Instead, it fails and sets **errno** to **EAGAIN**.

putmsg or **putpmsg** also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the stream, regardless of priority or whether **O_NONBLOCK** has been specified. No partial message is sent.

Return Values

On success, **putmsg** returns 0. On failure, **putmsg** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **putmsg** fails and sets **errno** to:

EAGAIN A non-priority message was specified, the **O_NONBLOCK** flag is set and the stream write queue is full due to internal flow control conditions.

EBADF *fd* is not a valid file descriptor open for writing.

EINTR A signal was caught during the **putmsg** system call.

EINVAL An undefined value was specified in *flags*, or *flags* is set to **RS_HIPRI** and no control part was supplied.

EINVAL The stream referenced by *fd* is linked below a multiplexor.

EINVAL For **putpmsg**, if *flags* is set to **MSG_HIPRI** and *band* is nonzero.

ENOSR Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

ENOSTR A stream is not associated with *fd*.

EIO A hangup condition was generated downstream for the specified stream, or the other end of the pipe is closed.

ERANGE The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

putmsg also fails if a STREAMS error message had been processed by the stream head before the call to **putmsg**. The error returned is the value contained in the STREAMS error message.

SEE ALSO

getmsg (BA_OS), **poll** (BA_OS), **putmsg** (BA_OS), **read** (BA_OS), **write** (BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

While one thread is blocked, siblings might still be executing.

NAME

pwrite – atomic position and write

SYNOPSIS

int pwrite(int *fd*, **char ***buf, **int** *nbytes*, **off_t** *offset*);

DESCRIPTION

The **pwrite** system call does an atomic position-and-write, eliminating the necessity of using a locking mechanism when both operations are desired and file descriptors are shared. **pwrite** is analogous to **write** but takes a fourth argument, *offset*. The write is done as if an **lseek** to *offset* (from the beginning of the file) were done first. Note that (though the semantics are analogous) an **lseek** is not actually performed; the file pointer is not affected by **pwrite**. The write of *nbytes* then starts at the specified offset.

The atomicity of **pwrite** enables processes or threads that share file descriptors to write to the shared file at a particular offset without using a locking mechanism that would be necessary to achieve the same result in separate **lseek** and **write** system calls. Atomicity is required as the file pointer is shared and one thread might move the pointer using *lseek* after another process completes an **lseek** but prior to the **write**.

Return Values

Upon successful completion, **pwrite** returns the number of bytes actually written from *buf*. Otherwise a -1 and an error is returned.

Errors

In the following conditions, **pwrite** fail and set **errno** to:

EAGAIN	Mandatory file/record locking is set, **O_NDELAY** or **O_NONBLOCK** is set, and there is a blocking record lock.
EAGAIN	Total amount of system memory available when reading via raw I/O is temporarily insufficient.
EAGAIN	An attempt is made to write to a stream that can not accept data with the **O_NDELAY** or **O_NONBLOCK** flag set.
EBADF	*fildes* is not a valid file descriptor open for writing.
EDEADLK	The **pwrite** was going to go to sleep and cause a deadlock to occur.
EFAULT	*buf* points outside the process's allocated address space.
EFBIG	An attempt is made to write a file that exceeds the process's file size limit or the maximum file size [see **ulimit**(BA_OS)].
EINTR	A signal was caught during the **pwrite** system call.
EINVAL	An attempt is made to write to a stream linked below a multiplexor.
EINVAL	The resulting file pointer would be negative. *fildes* is a remote file descriptor accessed using NFS, the Network File System, and the resulting file pointer would be negative.

EIO The process is in the background and is attempting to write to its controlling terminal whose **TOSTOP** flag is set; the process is neither ignoring nor blocking **SIGTTOU** signals, and the process group of the process is orphaned.

EIO *fildes* points to a device special file that is in the closing state.

ENOLCK The system record lock table was full, so the **pwrite** could not go to sleep until the blocking record lock was removed.

ENOLINK *fildes* is on a remote machine and the link to that machine is no longer active.

ENOSR An attempt is made to write to a stream with insufficient STREAMS memory resources available in the system.

ENOSPC During a **pwrite** to an ordinary file, there is no free space left on the device.

ENXIO The device associated with the file descriptor is a block-special or character-special file and the file-pointer value is out of range.

ERANGE An attempt is made to write to a stream with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.

ENOLCK Enforced record locking was enabled and **{LOCK_MAX}** regions are already locked in the system.

ESPIPE *fildes* is associated with a pipe or fifo.

ENOSYS The device for *fstype* does not support **lseek**.

SEE ALSO

creat(BA_OS), **fcntl**(BA_OS), **lseek**(BA_OS), **open**(BA_OS), **pread**(BA_OS), **write**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

While one thread is blocked, siblings might still be executing.

NAME

raise – send signal to program

SYNOPSIS

```
#include <signal.h>

int raise(int sig);
```

DESCRIPTION

`raise()` sends the signal *sig* to the executing program.

`raise()` returns zero if the operation succeeds. Otherwise, `raise()` returns `–1` and `errno` is set to indicate an error. `raise()` uses `kill()` to send the signal to the executing program:

```
kill(getpid(), sig);
```

[See kill(BA_OS) for a detailed list of failure conditions.]

ERRORS

Under the following conditions, the function `raise()` fails and sets `errno` to indicate an error.

`EINVAL` if *sig* is not a valid signal number.

SEE ALSO

getpid(BA_OS), kill(BA_OS), signal(BA_ENV).

LEVEL

Level 1.

NAME

read, readv – read from file

SYNOPSIS

```
#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);

#include <sys/types.h>
#include <sys/uio.h>

int readv(int fildes, struct iovec *iov, int iovcnt);
```

DESCRIPTION

read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. If *nbyte* is 0, **read** returns 0 and has no other results. *fildes* is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, **pipe**, or **ioctl** system call.

On devices capable of seeking, the **read** starts at a position in the file given by the file pointer associated with *fildes*. On return from **read**, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

readv performs the same action as **read**, but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], . . ., *iov*[*iovcnt*–1].

For **readv**, the **iovec** structure contains the following members:

```
void *    iov_base;
size_t    iov_len;
```

Each **iovec** entry specifies the base address and length of an area in memory where data snnni

In STREAMS message-nondiscard mode, **read** and **readv** retrieve data until they have read *nbyte* bytes, or until they reach a message boundary. If **read** or **readv** does not retrieve all the data in a message, the remaining data is replaced on the stream and can be retrieved by the next **read** or **readv** call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the **read** or **readv** returns is discarded, and is not available for a later **read**, **readv**, or **getmsg** [see **getmsg**(BA_OS)].

When attempting to read from a regular file with mandatory file/record locking set [see **chmod**(BA_OS)], and there is a write lock owned by another process on the segment of the file to be read:

If **O_NONBLOCK** is set, **read** returns –1 and sets **errno** to **EAGAIN**.

If **O_NONBLOCK** is clear, **read** sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

If no process has the pipe open for writing, **read** returns 0 to indicate end-of-file.

If some process has the pipe open for writing **read** returns 0.

If some process has the pipe open for writing and **O_NONBLOCK** is set, **read** returns –1 and sets **errno** to **EAGAIN**.

If **O_NONBLOCK** is clear, **read** blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

If **O_NONBLOCK** is set, **read** returns –1 and sets **errno** to **EAGAIN**.

If **O_NONBLOCK** is clear, **read** blocks until data becomes available.

When attempting to read a file associated with a stream that is not a pipe or FIFO, or terminal, and that has no data currently available:

If **O_NONBLOCK** is set, **read** returns –1 and sets **errno** to **EAGAIN**.

If **O_NONBLOCK** is clear, **read** blocks until data becomes available.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, **read** accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. **read** then returns the number of bytes read, and places the zero-byte message back on the stream to be retrieved by the next **read** or **getmsg** [see **getmsg**(BA_OS)]. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the stream. When a zero-byte message is read as the first message on a stream, a value of 0 is returned regardless of the **read** mode.

A **read** or **readv** from a STREAMS file returns the data in the message at the front of the stream head read queue, regardless of the priority band of the message.

Normally, a **read** from a STREAMS file can only process messages with data and without control information. The **read** fails if a message containing control information is encountered at the stream head. This default action can be changed by placing the stream in either control-data mode or control-discard mode with the **I_SRDOPT ioctl**(BA_OS). In control-data mode, control messages are converted to data messages by **read**. In control-discard mode, control messages are discarded by **read**, but any data associated with the control messages is returned to the user.

Return Values

On success, **read** and **readv** return a non-negative integer indicating the number of bytes actually read. On failure, **read** and **readv** return –1 and set **errno** to identify the error.

A **read** from a STREAMS file also fails if an error message is received at the stream head. In this case, **errno** is set to the value returned in the error message. If a hangup occurs on the stream being read, **read** continues to operate normally until the stream head read queue is empty. Thereafter, it returns 0.

Errors

In the following conditions, **read** and **readv** fail and set **errno** to:

EAGAIN
Mandatory file/record locking was set, **O_NONBLOCK** was set, and there was a blocking record lock.

EAGAIN
Total amount of system memory available when reading via raw I/O is temporarily insufficient.

EAGAIN
No data is waiting to be read on a file associated with a tty device and **O_NONBLOCK** was set.

EAGAIN
No message is waiting to be read on a stream and **O_NONBLOCK** was set.

EBADF *fildes* is not a valid file descriptor open for reading.

EBADMSG
Message waiting to be read on a stream is not a data message.

EDEADLK
The **read** was going to go to sleep and cause a deadlock to occur.

EINTR A signal was caught during the **read** or **readv** system call.

EINVAL
Attempted to read from a stream linked to a multiplexor.

EIO A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

EIO *fildes* is open to a device that is in the process of closing.

In addition, **readv** may return one of the following errors:

EINVAL
iovcnt was less than or equal to 0 or greater than 16.

EINVAL
The sum of the **iov_len** values in the *iov* array overflowed a 32-bit integer.

SEE ALSO

creat(BA_OS), **fcntl**(BA_OS), **getmsg**(BA_OS), **open**(BA_OS), **pread**(BA_OS), **streams**(BA_DEV), **types**(BA_ENV), **write**(BA_OS)

LEVEL

Level 1.

The enforcement mode of file and record locking has moved to Level 2 effective September 30, 1989.

NOTICES

read updates the time of last access [see **stat**(BA_OS)] of the file.

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

While one thread is blocked, siblings might still be executing.

NAME

readlink – read value of a symbolic link

SYNOPSIS

```
#include <unistd.h>

int readlink(const char *path, void *buf, size_t bufsiz);
```

DESCRIPTION

The function `readlink()` places the contents of the symbolic link referred to by *path* in the buffer *buf* which has size *bufsiz*. The contents of the link are not null-terminated when returned.

RETURN VALUE

Upon successful completion, the function `readlink()` returns the count of characters placed in the buffer; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `readlink()` fails, the buffer remains unchanged, and `errno` is set to:

`EACCES`	if search permission is denied for a component of the path prefix of *path*.
`EINVAL`	if *path* is not a symbolic link.
`EIO`	if an I/O error occurred while reading from or writing to the file system.
`ENOENT`	if the *path* does not exist.
`ELOOP`	if too many symbolic links are encountered in translating *path*.
`ENAMETOOLONG`	if the length of a *path* exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
`ENOSYS`	if this operation is not applicable for this file system type.

SEE ALSO

stat(BA_OS), symlink(BA_OS).

LEVEL

Level 1.

NAME

remove – remove file

SYNOPSIS

```
#include <stdio.h>

int remove(const char *path);
```

DESCRIPTION

The function `remove()` causes the file or empty directory whose name is the string pointed to by *path* to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless the file is created anew.

For files, `remove()` is identical to `unlink()`. For directories, `remove()` is identical to `rmdir()`.

RETURN VALUE

Upon successful completion, the function `remove()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `remove()` fails and sets `errno` to:

`EEXIST` if the directory to be removed contains directory entries other than `.` (the directory itself) and `..` (the parent directory).

`ENOTDIR` if a component of the path-prefix is not a directory.

`EACCES` if a component of the path-prefix denies search permission, or if write permission is denied on the parent directory of the directory or file to be removed.

`EBUSY` if the directory to be removed is currently in use by the system.

`EROFS` if the directory or file to be removed is located on a read-only file system.

`ELOOP` if too many symbolic names are encountered in translating *path*.

`ENAMETOOLONG`
if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

`ENOENT` if the *path* argument names a non-existent directory or points to an empty string.

`EPERM` if the named file is a directory and the effective user ID of the process does not have appropriate privileges.

SEE ALSO

rmdir(BA_OS), unlink(BA_OS).

LEVEL

Level 1.

NAME

rename – change the name of a file

SYNOPSIS

```
#include <unistd.h>

int rename(const char *old, const char *new);
```

DESCRIPTION

The function `rename()` changes the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new pathname of the file.

If the *old* argument and the *new* argument both refer to and link to the same existing file, the `rename()` function returns successfully and performs no other action.

If the *old* argument points to the pathname of a file that is not a directory, the *new* argument must not point to the pathname of a directory. If the link named by the *new* argument exists, it will be removed and *old* will be renamed to *new*. In this case, a link named *new* must remain visible to other processes throughout the renaming operation and will refer either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the pathname of a directory, the *new* argument must not point to the pathname of a file that is not a directory. If the directory named by the *new* argument exists, it will be removed and *old* will be renamed to *new*. In this case, a link named *new* will exist throughout the renaming operation and will refer either to the file referred to by *new* or *old* before the operation began. Thus, if *new* names an existing directory, it will be required to be an empty directory.

The *new* pathname must not contain a path prefix that names *old*. Write access permission is required for the directory containing *old* and the directory containing *new*. If the *old* argument points to the pathname of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link named by the *new* argument exists and the file's link count becomes zero when it is removed and no process has the file open, the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before `rename()` returns, but the removal of the file contents will be postponed until all references to the file have been closed.

Upon successful completion, the `rename()` function will mark for update the `st_ctime` and `st_mtime` fields of the parent directory of each file.

RETURN VALUE

Upon successful completion, the function `rename()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `rename()` fails and sets `errno` to:

EACCES	if a component of either path prefix denies search permission; or one of the directories containing *old* or *new* denies write permissions; or write permission is denied by a directory pointed to by the *old* or *new* parameters.
EBUSY	if the *new* is a directory and the mount point for a mounted file system.
EEXIST	if the link named by *new* is a directory containing entries other than . (the directory itself) and .. (the parent directory).
EINVAL	if *old* is a parent directory of *new*, or an attempt is made to rename . (the directory itself) or .. (the parent directory).
EISDIR	if the *new* parameter points to a directory but the *old* parameter points to a file that is not a directory.
ELOOP	if too many symbolic links were encountered in translating the pathname.
ENAMETOOLONG	if the length of a pathname exceeds {PATH_MAX}, or pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
ENOENT	if a component of either *old* or *new* does not exist, or the file referred to by either *old* or *new* does not exist, or either *old* or *new* point to an empty string.
ENOSPC	if the directory that would contain *new* cannot be extended.
ENOTDIR	if a component of either path prefix is not a directory; or the *old* parameter names a directory and the *new* parameter names a non-directory file.
EROFS	if the requested operation requires writing in a directory on a read-only file system.
EXDEV	if the links named by *old* and *new* are on different file systems.

SEE ALSO

link(BA_OS), unlink(BA_OS).

LEVEL

Level 1.

NAME

rmdir – remove a directory

SYNOPSIS

```
#include <unistd.h>

int rmdir(const char *path);
```

DESCRIPTION

The function `rmdir()` removes a directory.

The argument *path* specifies the path-name of the directory to be removed.

The directory must be empty, that is, not have any directory entries other than, `.` (the directory itself) and `..` (the parent directory).

If the directory's link count becomes zero and no process has the directory open, the space occupied by the directory is freed and the directory is no longer accessible. If one or more processes have the directory open when the last link is removed, the `.` and `..` entries, if present, are removed before `rmdir()` returns and no new entries may be created in the directory, but the directory is not removed until all references to the directory have been closed.

If path is a symbolic link, it is not followed.

Upon successful completion the function `rmdir()` marks for update the `st_ctime` and `st_mtime` fields of the parent directory.

RETURN VALUE

Upon successful completion, the function `rmdir()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `rmdir()` fails and sets `errno` to:

`EEXIST`	if the directory to be removed contains directory entries other than `.` (the directory itself) and `..` (the parent directory).
`ENOTDIR`	if a component of the path-prefix is not a directory.
`EACCES`	if a component of the path-prefix denies search permission, or if write permission is denied on the parent directory of the directory to be removed.
`EBUSY`	if the directory to be removed is currently in use by the system.
`EROFS`	if the directory to be removed is located on a read-only file system.
`EIO`	if a physical I/O error has occurred.
`ELOOP`	if too many symbolic links were encountered in translating path.
`ENAMETOOLONG`	if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
`ENOENT`	if the *path* argument names a non-existent directory or points to an empty string.

SEE ALSO

directory(BA_OS) mkdir(BA_OS)

LEVEL

Level 1.

NAME

seekdir – set position of directory stream

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>
void seekdir(DIR *dirp, long loc);
```

DESCRIPTION

The function `seekdir()` sets the position of the next `readdir()` operation on the directory stream specified by the *dirp* to the position specified by *loc*. The value of *loc* should have been returned from an earlier call to `telldir()`. The position reverts to the one associated with directory stream when the `telldir()` operation was performed.

SEE ALSO

directory(BA_OS), telldir(BA_OS).

LEVEL

Level 1.

NAME

setlocale – modifies and queries a program's locale

SYNOPSIS

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

DESCRIPTION

`setlocale()` selects the appropriate piece of the program's locale as specified by the *category* and *locale* arguments. The *category* argument may have the following values (defined in `<locale.h>`):

`LC_CTYPE` affects the behavior of the character handling functions (`isdigit()`, `tolower()`, *etc.*) and the multibyte character functions, `mbtowc()` and `wctomb()`.

`LC_NUMERIC` affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the non-monetary formatted information returned by `localeconv()`.

`LC_TIME` affects the behavior of time related functions, such as `getdate()` and `strftime()`.

`LC_COLLATE` affects the behavior of collating functions, such as `strcoll()` and `strxfrm()`.

`LC_MONETARY` affects the monetary formatted information returned by `localeconv()`.

`LC_MESSAGES` affects the behavior of message functions, such as `gettxt()`.

`LC_ALL` names the program's entire locale.

Each category corresponds to a set of databases which contain the relevant information for each defined locale. The location of a database is given by a path ending in `/usr/lib/`*locale*`/`*category*, where *locale* and *category* are the names of locale and category, respectively.

A value of `"C"` for the *locale* argument specifies the default environment.

A value of `""` for the *locale* argument specifies that the locale should be taken from environment variables. The order in which the environment variables are checked for the various categories is given below:

Category	1st Env. Var.	2nd
LC_CTYPE:	LC_CTYPE	LANG
LC_COLLATE:	LC_COLLATE	LANG
LC_TIME:	LC_TIME	LANG
LC_NUMERIC:	LC_NUMERIC	LANG
LC_MONETARY:	LC_MONETARY	LANG
LC_MESSAGES:	LC_MESSAGES	LANG

At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed. This has the effect of initializing each category to the locale described by the environment "C".

If a pointer to a string is given for *locale*, `setlocale()` attempts to set the locale for the given category to *locale*. If `setlocale()` succeeds, *locale* is returned. If `setlocale()` fails, a null pointer is returned and the program's locale is not changed.

For category `LC_ALL`, the behavior is slightly different. If a pointer to a string is given for *locale* and `LC_ALL` is given for *category*, `setlocale()` attempts to set the locale for all the categories to *locale*. The *locale* may be a simple locale, consisting of a single locale, or a composite locale. A composite locale is a string beginning with a `/` followed by the locale of each category separated by a `/`. If the locales for all the categories are the same after all the attempted locale changes, then `setlocale()` will return a pointer to the common simple locale. If there is a mixture of locales among the categories, then `setlocale()` will return a composite locale.

A null pointer for *locale* causes `setlocale()` to return the current locale associated with the *category*. The program's locale is not changed. If `LC_ALL` is given as the category and all the other categories do not have the same locale, then a composite locale is returned as above. If *category* is `LC_ALL` and the specified *locale* does not have files for all the categories (see table, above), `setlocale()` returns null.

SEE ALSO

conv(BA_LIB), ctime(BA_LIB), ctype(BA_LIB), getdate(BA_LIB), gettxt(BA_LIB), localeconv(BA_LIB), mbchar(BA_LIB), printf(BA_LIB), strcoll(BA_LIB), strftime(BA_LIB), strtod(BA_LIB), strxfrm(BA_LIB).

LEVEL

Level 1.

NAME

setpgid – set process group ID

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

int setpgid(pid_t pid, pid_t pgid);
```

DESCRIPTION

The function `setpgid()` is used to join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader will not change. Upon successful completion, the process group ID of the process with a process ID that matches *pid* will be set to *pgid*. As a special case, if *pid* is zero, the process ID of the calling process will be used. If *pgid* is zero the process ID of the indicated process will be used.

RETURN VALUE

Upon successful completion, the function `setpgid()` returns a value of `0`; otherwise, it returns a value of `–1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `setpgid()` fails and sets `errno` to:

`EACCES` if the value of the *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed an exec routine.

`EINVAL` if *pgid* is less than `(pid_t)0`, or greater than or equal to `{PID_MAX}`.

`EPERM` if the process indicated by the *pid* argument is a session leader.

`EPERM` if the value of the *pid* argument matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.

`EPERM` if the value of the *pgid* argument does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches the value of the *pgid* argument in the same session as the calling process.

`ESRCH` if the value of the *pid* argument does not match the process ID of the calling process or of a child process of the calling process.

SEE ALSO

exec(BA_OS), exit(BA_OS), fork(BA_OS), getpid(BA_OS), getpgid(BA_OS), setsid(BA_OS).

LEVEL

Level 1.

NAME

setsid – set session ID

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

pid_t setsid (void);
```

DESCRIPTION

The function `setsid()` sets the process group ID and session ID of the calling process to the process ID of the calling process, and releases the calling process's controlling terminal.

Upon returning, the calling process will be the session leader of a new session, will be the process group leader of a new process group, and will have no controlling terminal. The calling process will be the only process in the new process group and the only process in the new session.

RETURN VALUE

Upon successful completion, the function `setsid()` returns the calling process's session ID; otherwise, it returns a value of `(pid_t)-1` and sets `errno` to indicate an error.

ERRORS

Under the following condition, `setsid()` fails and sets `errno` to:

`EPERM` if the calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

USAGE

If the calling process is the last member of a pipeline started by a job-control shell, the shell may make the calling process a process group leader and the other processes of the pipeline members of that process group. In this case, the call to `setsid()` will fail. For this reason, a process that calls `setsid()` and expects to be part of a pipeline should always first fork; the parent should exit and the child should call `setsid()`. This will insure that the process will work reliably when started by both job-control shells and non-job control shells.

SEE ALSO

exec(BA_OS), exit(BA_OS), fork(BA_OS), getpid(BA_OS), getpgid(BA_OS), getsid(BA_OS), setpgid(BA_OS).

LEVEL

Level 1.

NAME

setuid, **setgid** – set user and group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

DESCRIPTION

The **setuid** system call sets the real user ID, effective user ID, and saved user **ID** of the calling process. The **setgid** system call sets the real group ID, effective group ID, and saved group ID of the calling process.

At login time, the real user ID, effective user ID, and saved user ID of the login process are set to the login ID of the user responsible for the creation of the process. The same is true for the real, effective, and saved group IDs; they are set to the group ID of the user responsible for the creation of the process.

When a process calls **exec(BA_OS)** to execute a file (program), the user and/or group identifiers associated with the process can change:

The real user and group IDs are always set to the real user and group IDs of the process calling **exec**.

The saved user and group IDs of the new process are always set to the effective user and group IDs of the process calling **exec**.

If the file executed is not a set-user-ID or set-group-ID file, the effective user and group IDs of the new process are set to the effective user and group IDs of the process calling **exec**.

If the file executed is a set-user-ID file, the effective user ID of the new process is set to the owner ID of the executed file.

If the file executed is a set-group-ID file, the effective group ID of the new process is set to the group ID of the executed file.

If the calling process has appropriate privileges, the real group ID, effective group ID and the saved set-group-ID are set to *gid*.

If the calling process does not have appropriate privileges, but its real group ID or saved set-group-ID is equal to *gid*, the effective group ID is set to *gid*; the real group ID and saved set-group-ID remain unchanged.

Return Values

On success, **setuid** and **setgid** return 0. On failure, **setuid** and **setgid** return –1 and set **errno** to identify the error.

Errors

In the following conditions, **setuid** and **setgid** fail and set **errno** to:

EPERM For **setuid**, the calling process does not have the appropriate privilege and the *uid* parameter does not match either the real or saved user IDs. For **setgid**, the calling process does not have the appropriate privilege and the *gid* parameter does not match either the real or saved group IDs.

EINVAL The *uid* or *gid* is out of range.

SEE ALSO

exec (BA_OS), **getgroups**(BA_OS), **getuid** (BA_OS), **stat**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

This ID number is an attribute of the containing process and is shared by sibling threads.

NAME

sigaction – detailed signal management

SYNOPSIS

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act,
    struct sigaction *oact);
```

DESCRIPTION

sigaction allows the calling process to examine and/or specify the action to be taken on delivery of a specific signal. [See **signal**(BA_ENV) for an explanation of general signal concepts.]

sig specifies the signal and can be assigned any of the signals specified in **signal**(BA_ENV), except **SIGKILL** and **SIGSTOP**.

If the argument *act* is not **NULL**, it points to a structure specifying the new action to be taken when delivering *sig*. If the argument *oact* is not **NULL**, it points to a structure where the action previously associated with *sig* is to be stored on return from **sigaction**.

The **sigaction** structure includes the following members:

```
void        (*sa_handler)();
sigset_t    sa_mask;
int         sa_flags;
```

sa_handler specifies the disposition of the signal and may take any of the values specified in **signal**(BA_OS).

sa_mask specifies a set of signals to be blocked while the signal handler is active. On entry to the signal handler, that set of signals is added to the set of signals already being blocked when the signal is delivered. In addition, the signal that caused the handler to be executed will also be blocked, unless the **SA_NODEFER** flag has been specified. **SIGSTOP** and **SIGKILL** cannot be blocked (the system silently enforces this restriction).

sa_flags specifies a set of flags used to modify the delivery of the signal. It is formed by a logical **OR** of any of the following values:

SA_ONSTACK If set and the signal is caught and an alternate signal stack has been declared the signal is delivered to the calling process on that stack. Otherwise, the signal should be delivered on the current stack.

Alternate signal handling stacks can be defined via the **sigaltstack**(BA_OS) system call.

SA_RESETHAND If set and the signal is caught, the disposition of the signal is reset to **SIG_DFL** and the signal will not be blocked on entry to the signal handler (**SIGILL**, **SIGTRAP**, and **SIGPWR** cannot be automatically reset when delivered; the system silently enforces this restriction).

SA_RESTART If set and the signal is caught, a system call that is interrupted by the execution of this signal's handler is transparently restarted by the system. Otherwise, that system call returns an **EINTR** error. Not all system calls can be restarted, for example, **sleep**(2) and **pause**(2) cannot be restarted.

SA_SIGINFO If cleared and the signal is caught, *sig* is passed as the only argument to the signal-catching function. If set and the signal is caught, two additional arguments are passed to the signal-catching function. If the second argument is not equal to **NULL**, it points to a **siginfo_t** structure containing the reason why the signal was generated the third argument points to a **ucontext_t** structure containing the receiving process's context when the signal was delivered

SA_NOCLDWAIT If set and *sig* equals **SIGCHLD**, the system will not create zombie processes when children of the calling process exit. If the calling process subsequently issues a **wait**(BA_OS), it blocks until all of the calling process's child processes terminate, and then returns a value of **–1** with **errno** set to **ECHILD**.

SA_NOCLDSTOP If set and *sig* equals **SIGCHLD**, *sig* will not be sent to the calling process when its child processes stop or continue. underlying execution entities kernel execution entities

Return Values

On success, **sigaction** returns 0. On failure, **sigaction** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **sigaction** fails and sets **errno** to:

EINVAL The value of the *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

EFAULT *act* or *oact* points outside the process's allocated address space.

SEE ALSO

exit(BA_OS), **kill**(BU_CMD), **kill**(BA_OS), **pause**(BA_OS), **sigaltstack**(BA_OS), **signal**(BA_OS), **signal**(BA_ENV), **sigprocmask**(BA_OS), **sigsend**(BA_OS), **sigsetops**(BA_OS), **sigsuspend**(BA_OS), **wait**(BA_OS)

LEVEL

Level 1.

NOTICES

If the system call is reading from or writing to a terminal and the terminal's **NOFLSH** bit is cleared, data may be flushed.

Considerations for Threads Programming

The handler defined by *act* is common to all threads in a process.

The Threads Library does not support alternate signal handling stacks for threads.

The **SA_WAITSIG** flag (see description above) can be used in support of threads libraries.

Further details can be found in **signal**(BA_ENV).

NAME

sigaltstack – set or get signal alternate stack context

SYNOPSIS

```
#include <signal.h>

int sigaltstack(const stack_t *ss, stack_t *oss);
```

DESCRIPTION

sigaltstack allows users to define an alternate stack area on which signals are to be processed. If *ss* is non-zero, it specifies a pointer to, and the size of a stack area on which to deliver signals, and tells the system if the process is currently executing on that stack. When a signal's action indicates its handler should execute on the alternate signal stack [specified with a **sigaction**(2) call], the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler's execution.

The structure **sigaltstack** includes the following members.

```
char  *ss_sp
int    ss_size
int    ss_flags
```

If *ss* is not **NULL**, it points to a structure specifying the alternate signal stack that will take effect upon return from **sigaltstack**. The **ss_sp** and **ss_size** fields specify the new base and size of the stack, which is automatically adjusted for direction of growth and alignment. The **ss_flags** field specifies the new stack state and may be set to the following:

SS_DISABLE The stack is to be disabled and **ss_sp** and **ss_size** are ignored. If **SS_DISABLE** is not set, the stack will be enabled. **SS_DISABLE** is the only way users can disable the alternate signal stack.

If *oss* is not **NULL**, it points to a structure specifying the alternate signal stack that was in effect prior to the call to **sigaltstack**. The **ss_sp** and **ss_size** fields specify the base and size of that stack. The **ss_flags** field specifies the stack's state, and may contain the following values:

SS_ONSTACK The process is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the process is executing on it will fail. **SS_ONSTACK** cannot be modified by users.

SS_DISABLE The alternate signal stack is currently disabled.

Return Values

On success, **sigaltstack** returns 0. On failure, **sigaltstack** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **sigaltstack** fails and sets **errno** to:

EFAULT Either *ss* or *oss* points outside the process's allocated address space.

EINVAL *ss* is non-null and the **ss_flags** field pointed to by *ss* contains invalid flags. The only flag considered valid is **SS_DISABLE**.

EPERM An attempt was made to modify an active stack.

ENOMEM The size of the alternate stack area is less than **MINSIGSTKSZ**.

USAGE

The value **SIGSTKSZ** is defined to be the number of bytes that would be used to cover the usual case when allocating an alternate stack area. The value **MINSIGSTKSZ** is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, a program should add that amount to its stack requirements to allow for the operating system overhead.

The following code fragment is typically used to allocate an alternate stack.

```
if ((sigstk.ss_sp = (char *)malloc(SIGSTKSZ)) == NULL)
     /* error return */;

sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, (stack_t *)0) < 0)
     perror("sigaltstack");
```

SEE ALSO

getcontext(BA_OS), **sigaction**(BA_OS),

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

The Threads Library does not support alternate signal handling stacks for threads. See **signal**(BA_OS) for further details.

NAME

signal, **sigset**, **sighold**, **sigrelse**, **sigignore**, **sigpause** – simplified signal management

SYNOPSIS

```
#include <signal.h>

void (*signal(int sig, void (*disp)(int)))(int);

void (*sigset(int sig, void (*disp)(int)))(int);

int sighold(int sig);

int sigrelse(int sig);

int sigignore(int sig);

int sigpause(int sig);
```

DESCRIPTION

These functions provide simplified signal management for application processes. See **signal**(BA_OS) for an explanation of general signal concepts.

signal and **sigset** are used to modify signal dispositions. *sig* specifies the signal, which may be any signal except **SIGKILL** and **SIGSTOP**. *disp* specifies the signal's disposition, which may be **SIG_DFL**, **SIG_IGN**, or the address of a signal handler. If **signal** is used, *disp* is the address of a signal handler, and *sig* is not **SIGILL**, **SIGTRAP**, or **SIGPWR**, the system first sets the signal's disposition to **SIG_DFL** before executing the signal handler. If **sigset** is used and *disp* is the address of a signal handler, the system adds *sig* to the calling process's signal mask before executing the signal handler; when the signal handler returns, the system restores the calling process's signal mask to its state prior to the delivery of the signal. In addition, if **sigset** is used and *disp* is equal to **SIG_HOLD**, *sig* is added to the calling process's signal mask and the signal's disposition remains unchanged. However, if **sigset** is used and *disp* is not equal to **SIG_HOLD**, *sig* will be removed from the calling process's signal mask.

sighold adds *sig* to the calling process's signal mask.

sigrelse removes *sig* from the calling process's signal mask.

sigignore sets the disposition of *sig* to **SIG_IGN**.

sigpause removes *sig* from the calling process's signal mask and suspends the calling process until a signal is received.

Return Values

On success, **signal** returns the signal's previous disposition. On failure, **signal** returns **SIG_ERR** and sets **errno** to identify the error.

Errors

In the following conditions, this function fails and set **errno** to:

EINVAL The value of the *sig* argument is not a valid signal or is equal to **SIGKILL** or **SIGSTOP**.

EINTR A signal was caught during the system call **sigpause**.

USAGE

If **signal** is used to set **SIGCHLD**'s disposition to a signal handler, **SIGCHLD** will not be sent when the calling process's children are stopped or continued.

If any of the above functions are used to set **SIGCHLD**'s disposition to **SIG_IGN**, the calling process's child processes will not create zombie processes when they terminate. If the calling process subsequently waits for its children, it blocks until all of its children terminate; it then returns a value of –1 with **errno** set to **ECHILD**. [see **wait**(BA_OS), **waitid**(BA_OS)].

SEE ALSO

kill(BA_OS), **pause**(BA_OS), **sigaction**(BA_OS), **signal**(BA_ENV), **sigsend**(BA_OS), **wait**(BA_OS), **waitid**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Signal dispositions (that is, default/ignore/handler) are a process attribute and are shared by all threads. Signal masks, on the other hand, are maintained independently per thread.

NAME

sigpending – examine signals that are blocked and pending

SYNOPSIS

```
#include <signal.h>

int sigpending(sigset_t *set);
```

DESCRIPTION

The **sigpending** function retrieves those signals that have been sent to the calling process but are being blocked from delivery by the calling process's signal mask. The signals are stored in the space pointed to by the argument *set*.

Return Values

On success, **sigpending** returns 0. On failure, **sigpending** returns –1 and sets **errno** to identify the error.

SEE ALSO

sigaction(BA_OS), **sigprocmask**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

The set returned is the union of

Signals pending to the calling thread but blocked by that thread's signal mask.

Signals pending to the process but blocked by every currently running thread in the process.

In general, the status from **sigpending** is only advisory. A signal pending to the containing process might be delivered to a sibling thread (if any become eligible) after the return of this system call. See **signal**(BA_ENV) for further details.

NAME

sigprocmask – change or examine signal mask

SYNOPSIS

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

DESCRIPTION

The **sigprocmask** function is used to examine and/or change the calling process's signal mask. If the value is **SIG_BLOCK**, the set pointed to by the argument *set* is added to the current signal mask. If the value is **SIG_UNBLOCK**, the set pointed by the argument *set* is removed from the current signal mask. If the value is **SIG_SETMASK**, the current signal mask is replaced by the set pointed to by the argument *set*. If the argument *oset* is not **NULL**, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is **NULL**, the value *how* is not significant and the process's signal mask is unchanged; thus, the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to **sigprocmask**, at least one of those signals will be delivered before the call to **sigprocmask** returns.

It is not possible to block those signals that cannot be ignored [see **sigaction**(BA_OS)]. This restriction is silently imposed by the system.

If **sigprocmask** fails, the process's signal mask is not changed.

Return Values

On success, **sigprocmask** returns 0. On failure, **sigprocmask** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **sigprocmask** fails and sets **errno** to:

EINVAL The value of the *how* argument is not equal to one of the defined values.

EFAULT The value of *set* or *oset* points outside the process's allocated address space.

SEE ALSO

sigaction(BA_OS), **signal**(BA_OS), **sigsetops**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Signal masks are maintained per thread. See **signal**(BA_OS) for further details.

NAME

sigsend, **sigsendset** – send a signal to a process or a group of processes

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>
#include <sys/procset.h>

int sigsend(idtype_t idtype, id_t id, int sig);

int sigsendset(const procset_t *psp, int sig);
```

DESCRIPTION

sigsend sends a signal to the process or group of processes specified by *id* and *idtype*. The signal to be sent is specified by *sig* and is either zero or one of the values listed in **signal**(BA_OS). If *sig* is zero (the null signal), error checking is performed but no signal is actually sent. This value can be used to check the validity of *id* and *idtype*.

In order to send the signal to the target process (*pid*), the sending process must have permission to do so, subject to the following ownership restrictions:

> The real or effective user ID of the sending process must match the real or saved [from exec(BA_OS)] user ID of the receiving process, unless the sending process has the **P_OWNER** privilege, or *sig* is **SIGCONT** and the sending process has the same session ID as the receiving process.

If *idtype* is **P_PID**, *sig* is sent to the process with process ID *id*.

If *idtype* is **P_PGID**, *sig* is sent to any process with process group ID *id*.

If *idtype* is **P_SID**, *sig* is sent to any process with session ID *id*.

If *idtype* is **P_UID**, *sig* is sent to any process with effective user ID *id*.

If *idtype* is **P_GID**, *sig* is sent to any process with effective group ID *id*.

If *idtype* is **P_CID**, *sig* is sent to any process with scheduler class ID *id* [see **priocntl**(KE_OS)].

If *idtype* is **P_ALL**, *sig* is sent to all processes and *id* is ignored.

If *id* is **P_MYID**, the value of *id* is taken from the calling process.

The process with a process ID of 0 is always excluded. The process with a process ID of 1 is excluded unless *idtype* is equal to **P_PID**.

sigsendset provides an alternate interface for sending signals to sets of processes. This function sends signals to the set of processes specified by *psp*. *psp* is a pointer to a structure of type **procset_t**, defined in **sys/procset.h**, which includes the following members:

```
idop_t      p_op;
idtype_t    p_lidtype;
id_t        p_lid;
idtype_t    p_ridtype;
id_t        p_rid;
```

p_lidtype and **p_lid** specify the ID type and ID of one (''left'') set of processes; **p_ridtype** and **p_rid** specify the ID type and ID of a second (''right'') set of processes. ID types and IDs are specified just as for the *idtype* and *id* arguments to **sigsend**. **p_op** specifies the operation to be performed on the two sets of processes to get the set of processes the system call is to apply to. The valid values for **p_op** and the processes they specify are:

POP_DIFF set difference: processes in left set and not in right set

POP_AND set intersection: processes in both left and right sets

POP_OR set union: processes in either left or right set or both

POP_XOR set exclusive-or: processes in left or right set but not in both

Return Values

On success, **sigsend** and **sigsendset** return 0. On failure, **sigsend** and **sigsendset** return –1 and set **errno** to identify the error.

Errors

In the following conditions, **sigsend** and **sigsendset** fail and set **errno** to:

EINVAL *sig* is not a valid signal number.

EINVAL *idtype* is not a valid idtype field.

EPERM The calling process does not have the appropriate privilege, the real or effective user ID of the sending process does not match the real or effective user ID of the receiving process, and the calling process is not sending **SIGCONT** to a process that shares the same session.

ESRCH No process can be found corresponding to that specified by *id* and *idtype*.

In addition, **sigsendset** fails if:

EFAULT **psp** points outside the process's allocated address space.

SEE ALSO

kill(BA_OS), **priocntl**(KE_OS), **signal**(BA_OS),

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Signals can be posted from one process to the designated processes via the **sigsend** system call but not to specific threads within those processes. See **signal**(BA_OS) for further details. See **thr_kill**(MT_LIB) for details of intra-process signaling between threads.

NAME

sigsetops: sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate sets of signals.

SYNOPSIS

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);

int sigismember(const sigset_t *set, int signo);
```

DESCRIPTION

The above primitives manipulate the `sigset_t` data types, representing the sets of signals supported by the implementation. Examples of sets of signals known to the system are the set blocked from delivery to a process and the set pending a process.

The `sigemptyset()` function excludes all signals from the set pointed to by the argument *set*. The `sigfillset()` function initializes the set pointed to by the argument *set* so that all signals are included. The `sigaddset()` and `sigdelset()` functions respectively add and delete the individual signal specified by the value of the argument *signo* from the set pointed to by the argument *set*. The `sigismember()` function checks whether the signal specified by the value of the argument *signo* is a member of the set pointed to by the argument *set*.

Any object of type `sigset_t` must be initialized by applying either `sigemptyset()` or `sigfillset()` before applying any other operation.

RETURN VALUE

Upon successful completion, the function `sigismember()` returns a value of `1` if the specified signal is a member of the specified set, or a value of `0` if it is not. Upon successful completion, the other functions return a value of `0`; otherwise, they return a value of `–1` and sets `errno` to indicate an error.

ERRORS

Under the following condition, the functions `sigsetops()`, `sigaddset()`, `sigdelset()`, and `sigismember()` fail and set `errno` to:

`EINVAL` if the value of the *signo* argument is not a valid signal.

SEE ALSO

sigaction(BA_OS), signal(BA_ENV), sigprocmask(BA_OS), sigpending(BA_OS), sigsuspend(BA_OS).

LEVEL

Level 1.

NAME

sigsuspend – install a signal mask and suspend process until signal

SYNOPSIS

```
#include <signal.h>

int sigsuspend(const sigset_t *set);
```

DESCRIPTION

sigsuspend replaces the process's signal mask with the set of signals pointed to by the argument *set* and then suspends the process until delivery of a signal whose action is either to execute a signal catching function or to terminate the process.

If the action is to terminate the process, **sigsuspend** does not return. If the action is to execute a signal catching function, **sigsuspend** returns after the signal catching function returns. On return, the signal mask is restored to the set that existed before the call to **sigsuspend**.

It is not possible to block those signals that cannot be ignored [see **signal**(BA_OS)]; this restriction is silently imposed by the system.

Return Values

Because **sigsuspend** suspends process execution indefinitely, there is no successful return value. On failure, **sigsuspend** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **sigsuspend** fails and sets **errno** to:

EINTR A signal is caught by the calling process and control is returned from the signal catching function.

SEE ALSO

signal(BA_OS), **sigprocmask**(BA_OS), **sigsetops**(BA_OS),

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

In multithreaded programs, signal masks are defined per thread. See **signal**(BA_OS) for further details.

While one thread is blocked, siblings might still be executing.

NAME

sigwait – wait for a signal to be posted

SYNOPSIS

```
#include <signal.h>

int sigwait(sigset_t *set);
```

DESCRIPTION

This function atomically chooses and clears a pending masked signal from *set* and returns the number of the signal chosen. If no signal in *set* is pending at the time of the call, the calling function shall be suspended until one or more signals become pending. This suspension is indefinite in extent.

The *set* of signals remains blocked after return.

An application should not mix use of **sigwait** and **sigaction** for a given signal number because the results may be unpredictable.

Return Values

Upon successful completion, **sigwait** returns the signal number of the received signal. Otherwise, a negative value is returned and **errno** is set to indicate the error.

Errors

If any of the following conditions occurs, **sigwait** returns a negative value and sets **errno** to the corresponding value:

EINVAL *set* contains an invalid or unsupported signal number

EFAULT *set* points to an illegal address.

SEE ALSO

kill(BA_OS), sigaction(BA_OS), signal(BA_ENV), sigpending(BA_OS), sigsend(BA_OS), sigsuspend(BA_OS)

NOTICES

Considerations for Threads Programming

The **sigwait** system call allows a multithreaded application to use a synchronous organization for signal handling.

Usage

The semantics of **sigwait** make it ideal for a thread that will be dedicated to handling certain signal types for a process. The functionality that might have been placed in a separate handler function could be placed after the return from **sigwait** to be executed once a signal arrives. Once handling is complete, the thread could call **sigwait** again to block itself until arrival of the next signal.

To be sure that signals are delivered to the intended thread:

All threads in the process (including the thread that will be using **sigwait**) should mask the relevant signal numbers.

Multiple **sigwait** system calls for a given signal number compete for each single delivery of that signal number.

No thread should define a handler function for those signal numbers.

See **signal**(BA_ENV) for further details.

Code to handle a signal type on return from **sigwait** is not considered a handler in the containing process' disposition for that signal type. It is important that signal types handled by a thread using **sigwait**(BA_OS) be included in the signal mask of *every thread*, otherwise, the default response for the process will be triggered. Even the thread calling **sigwait** must mask that signal type because a signal of that type may arrive while the thread is between calls to **sigwait**(BA_OS).

While one thread is blocked, siblings might still be executing.

sigwait for signals that are normally synchronously generated (e.g. **SIGFPE**) will not return because the waiting thread cannot execute code that will generate that fault. However, an externally and/or asynchronously, generated **SIGFPE** would cause a waiting thread to return.

LEVEL

Level 1.

NAME

sleep – suspend execution for interval

SYNOPSIS

```
#include <unistd.h>

unsigned sleep(unsigned seconds);
```

DESCRIPTION

The function `sleep()` suspends the current process from execution for the number of seconds specified by the argument *seconds*. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals (on the second, according to an internal clock) and (2) because any signal caught will terminate the `sleep()` following execution of that signal-catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system.

The current process is suspended by calling the `alarm()` function [see alarm(BA_OS)] and pausing until the `SIGALRM` signal (or some other signal) is delivered. The previous disposition of the `SIGALRM` signal is saved before calling `alarm()`, and restored before returning from `sleep()`. If the calling process had set up an alarm before calling `sleep()`, and if the argument *seconds* exceeds the time left until that alarm would expire, the process sleeps only until the original alarm expires.

RETURN VALUE

The function `sleep()` returns the unslept amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested suspension time or premature arousal due to another caught signal. The function `sleep()` is always successful.

SEE ALSO

alarm(BA_OS), pause(BA_OS), sigaction(BA_OS), signal(BA_OS), signal(BA_ENV), sigsetjmp(BA_LIB), sigsuspend(BA_OS).

LEVEL

Level 1.

NAME

stat, **lstat**, **fstat** – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *path, struct stat *buf);

int lstat(const char *path, struct stat *buf);

int fstat(int fildes, struct stat *buf);
```

DESCRIPTION

The **stat** system calls get information about a file. *path* points to a pathname naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

Note that in a Remote File Sharing environment, the information returned by **stat** depends on the user/group mapping set up between the local and remote computers. [See **idload**(RS_CMD)]

lstat obtains file attributes similar to **stat**, except when the named file is a symbolic link; in that case **lstat** returns information about the link, while **stat** returns information about the file the link references.

fstat obtains information about an open file known by the file descriptor *fildes*, obtained from a successful **creat**, **open**, **dup**, **fcntl**, **pipe**, or **ioctl** system call.

buf is a pointer to a **stat** structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* includes the following members:

```
mode_t  st_mode;    /* File mode [see mknod)] */
ino_t   st_ino;     /* Inode number */
dev_t   st_dev;     /* ID of device containing */
                    /* a directory entry for this file */
dev_t   st_rdev;    /* ID of device */
                    /* This entry is defined only for */
                    /* char special or block special files */
nlink_t st_nlink;   /* Number of links */
uid_t   st_uid;     /* User ID of the file's owner */
gid_t   st_gid;     /* Group ID of the file's group */
off_t   st_size;    /* File size in bytes */
time_t  st_atime;   /* Time of last access */
time_t  st_mtime;   /* Time of last data modification */
time_t  st_ctime;   /* Time of last file status change */
                    /* Times measured in seconds since */
                    /* 00:00:00 UTC, Jan. 1, 1970 */
long    st_blksize; /* Preferred I/O block size */
long    st_blocks;  /* Number of 512 blocks allocated */
                    /*A files residing on an s5*/
                    /*file system reports number of*/
                    /*blocks allocated assuming no*/
                    /*holes in the file*/
```

st_mode The mode of the file as described in **mknod**(1M). In addition to the modes described in **mknod**(1M), the mode of a file may also be **S_IFLNK** if the file is a symbolic link. (Note that **S_IFLNK** may only be returned by **lstat**.)

st_ino This field uniquely identifies the file in a given file system. The pair **st_ino** and **st_dev** uniquely identifies regular files.

st_dev This field uniquely identifies the file system that contains the file. Its value may be used as input to the **ustat** system call to determine more information about this file system. No other meaning is associated with this value.

st_rdev This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.

st_nlink This field should be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For regular files, this is the address of the end of the file. Defined for block devices, although the size may be zero if the device size is unknown. See also **pipe**(BA_OS).

st_atime Time when file data was last accessed. Changed by the following system calls: **creat**, **mknod**, **pipe**, **utime**, and **read**.

st_mtime Time when data was last modified. Changed by the following system calls: **creat**, **mknod**, **pipe**, **utime**, and **write**.

st_ctime Time when file status was last changed. Changed by the following system calls: **chmod**, **chown**, **creat**, **link**, **mknod**, **pipe**, **unlink**, **utime**, and **write**.

st_blksize A hint as to the ''best'' unit size for I/O operations. This field is not defined for block-special or character-special files.

st_blocks The total number of physical blocks of size 512 bytes actually allocated on disk. This field is not defined for block-special or character-special files. A file residing on an s5 filesystem reports number of blocks allocated assuming no holes in the file.

Return Values

On success, **stat**, **lstat**, and **fstat** return 0. On failure, **stat**, **lstat**, and **fstat** return –1 and set **errno** to identify the error.

Errors

In the following conditions, **stat** and **lstat** fail and set **errno** to:

EACCES Search permission is denied for a component of the path prefix.

EACCES Read permission is denied on the named file.

ELOOP Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.

ENOENT The named file does not exist or is the null pathname.

ENOTDIR A component of the path prefix is not a directory.

In the following conditions, **fstat** fails and sets **errno** to:

EBADF *fildes* is not a valid open file descriptor.

SEE ALSO

chmod (BA_OS), **chown** (BA_OS), **creat** (BA_OS), **fattach** (BA_LIB), **link** (BA_OS), **mknod** (BA_OS), **pipe** (BA_OS), **read** (BA_OS), **stat** (BA_OS), **time** (SD_CMD), **unlink** (BA_OS), **utime** (BA_OS), **write** (BA_OS)

LEVEL

Level 1.

NAME

statvfs, fstatvfs – get file system information

SYNOPSIS

```
#include <sys/types.h>
#include <sys/statvfs.h>

int statvfs(const char *path, struct statvfs *buf);

int fstatvfs(int fildes, struct statvfs *buf);
```

DESCRIPTION

The function `statvfs()` returns descriptive information about a mounted file system containing the file referenced by *path*. *buf* is a pointer to a structure (described below) which will be filled by the system call.

path must name a file which resides on the file system. The file system type must be known to the operating system. Read, write, or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

The `statvfs()` structure pointed to by *buf* includes the following members:

```
ulong  f_bsize;    /* preferred file system block size */
ulong  f_frsize;   /* fundamental file system block size
                        (if supported) */
ulong  f_blocks;   /* total # of blocks of f_frsize
                        on file system */
ulong  f_bfree;    /* total # of free blocks */
ulong  f_bavail;   /* # of free blocks avail to non-super-user */
ulong  f_files;    /* total # of file nodes (inodes) */
ulong  f_ffree;    /* total # of free file nodes */
ulong  f_favail;   /* # of file nodes (inodes) avail to
                        non-super-user */
ulong  f_fsid;          /* file system id */
char   f_basetype[FSTYPSZ];  /* target fs type name,
                        null-terminated */
ulong  f_flag;     /* bit mask of flags */
ulong  f_namemax;  /* maximum filenamelength */
char   f_fstr[32];/* file system specific string */
```

`f_basetype` contains a null-terminated file system type name. The constant `FSTYPSZ` is defined in the header file `<statvfs.h>`.

The following flags can be returned in the `f_flag` field:

```
ST_RDONLY        /* read-only file system */
ST_NOSUID        /* does not support setuid/setgid semantics */
```

Similarly, the function `fstatvfs()` obtains information about a mounted file system containing the file referenced by *fildes*.

RETURN VALUE

Upon successful completion, the function `statvfs()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `statvfs()` fails and sets `errno` to:

`EACCES` if search permission is denied on a component of the *path* prefix.

`ELOOP` if too many symbolic links were encountered in translating *path*.

`ENAMETOOLONG`
if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

`ENOENT` if the file referred to by *path* does not exist.

`ENOTDIR` if a component of the path prefix of *path* is not a directory.

The function `fstatvfs()` fails and sets `errno` to:

`EBADF` if *fildes* is not an open file descriptor.

SEE ALSO

chmod(BA_OS), chown(BA_OS), creat(BA_OS), dup(BA_OS), fcntl(BA_OS), link(BA_OS), mknod(BA_OS), open(BA_OS), pipe(BA_OS), read(BA_OS), time(BA_OS), unlink(BA_OS), ustat(BA_OS), utime(BA_OS), write(BA_OS).

LEVEL

Level 1.

NAME

stime – set time

SYNOPSIS

```
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

int stime(const time_t *tp);
```

DESCRIPTION

stime sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 UTC January 1, 1970.

Return Values

On success, **stime** returns 0. On failure, **stime** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **stime** fails and sets **errno** to:

EPERM The calling process does not have the appropriate privilege

SEE ALSO

time (SD_CMD)

LEVEL

Level 1.

NAME

symlink – make symbolic link to a file

SYNOPSIS

```
int symlink(const char *path1, const char *path2);
```

DESCRIPTION

A symbolic link *path2* is created to *path1* (*path2* is the name of the file created, *path1* is the pathname used to create the symbolic link). Either name may be an arbitrary pathname and *path1* need not exist; the files need not be on the same file system.

The file to which the symbolic link points is used when an `open()` [see open(BA_OS)] operation is performed on the link.

RETURN VALUE

Upon successful completion, the function `symlink()` returns a value of zero; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `symlink()` fails and sets `errno` to:

`EACCESS` if write permission is denied in the directory where the symbolic link is being created.

`ENOTDIR` if a component of the path prefix of *path2* is not a directory.

`ENAMETOOLONG` if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

`ENOENT` if a component of the path prefix of *path2* does not exist.

`EACCES` if search permission is denied for a component of the path prefix of *path2*.

`ELOOP` if too many symbolic links are encountered in translating *path2*.

`EEXIST` if the file referred to by *path2* already exists.

`EROFS` if the file *path2* would reside on a read-only file system.

`ENOSPC` if the directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the file system containing the directory.

`ENOSPC` if the new symbolic link cannot be created because no space is left on the file system which will contain the link.

`ENOSPC` if no free inodes are on the file system on which the file is being created.

`ENOSYS` if this operation is not applicable for this file system type.

USAGE

A `stat()` on a symbolic link returns the linked-to file, while an `lstat()` returns information about the link itself [see stat(BA_OS)]. This can lead to unexpected results when a symbolic link is made to a directory. To avoid confusion in programs, the `readlink()` call can be used to read the contents of a symbolic link [see readlink(BA_OS)].

SEE ALSO

link(BA_OS), readlink(BA_OS), stat(BA_OS), unlink(BA_OS).

LEVEL

Level 1.

NAME

sync – update super-block

SYNOPSIS

```
void sync(void);
```

DESCRIPTION

The function `sync()` causes all information in transient memory that updates a file system to be written out to the file system. This includes modified super-blocks, modified i-nodes, and delayed block I/O.

The function `sync()` should be used by programs which examine a file system.

The writing, although scheduled, is not necessarily complete upon return from the function `sync()`.

USAGE

The function `sync()` is not recommended for use by application-programs.

SEE ALSO

fsync(BA_OS).

LEVEL

Level 1.

NAME

sysconf – get configurable system variables

SYNOPSIS

```
#include <unistd.h>

long sysconf(int name);
```

DESCRIPTION

The `sysconf()` function provides a method for the application to determine the current value of a configurable system limit or option (variable).

The *name* argument represents the system variable to be queried. The following table lists the minimal set of system variables from `<limits.h>`, `<unistd.h>` or `<time.h>` (for `CLK_TCK`) that can be returned by `sysconf()`, and the symbolic constants, defined in `<unistd.h>` that are the corresponding values used for *name.*

Variable	Value of *name*
ARG_MAX	_SC_ARG_MAX
CHILD_MAX	_SC_CHILD_MAX
CLK_TCK	_SC_CLK_TCK
NGROUPS_MAX	_SC_NGROUPS_MAX
OPEN_MAX	_SC_OPEN_MAX
PASS_MAX	_SC_PASS_MAX
PAGESIZE	_SC_PAGESIZE
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_VERSION	_SC_VERSION
_XOPEN_VERSION	_SC_XOPEN_VERSION

The value of `CLK_TCK` may be variable and it should not be assumed that `CLK_TCK` is a compile-time constant. The value of `CLK_TCK` is the same as the value of `sysconf(_SC_CLK_TCK)`.

`sysconf` can also return the following values:

Name	Return Value
_SC_NPROCESSORS_CONF	Number of configured processors
_SC_NPROCESSORS_ONLN	Number of online processors
_SC_NPROCESSES	

RETURN VALUE

Upon successful completion, the function `sysconf()` returns the current variable value on the system. The value returned will not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>` or `<unistd.h>`. The value will not change during the lifetime of the calling process. If *name* is an invalid value, `sysconf()` will return `-1` and set `errno` to indicate the error. If `sysconf()` fails due to a value of

name that is not defined on the system, the function will return a value of `-1` without changing the value of `errno`. Additionally, a call to `setrlimit()` may cause the value of `OPEN_MAX` to change.

ERRORS

Under the following condition, the function `sysconf()` fails and sets `errno` to:

`EINVAL` if the value of the argument *name* is invalid.

SEE ALSO

fpathconf(BA_OS).

LEVEL

Level 1.

NAME

system – issue a command

SYNOPSIS

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

int system(const char *string);
```

DESCRIPTION

The function `system()` causes the argument *string* to be given as input to a command interpreter and execution process. That is, the argument *string* is interpreted as a command, and then the command is executed. A null pointer may be used for *string* to inquire whether a command processor exists.

Commands

A *blank* is a tab or a space.

A *word* is a sequence of characters excluding blanks.

A *parameter name* is a sequence of letters, digits, or underscores beginning with a letter or underscore. A *parameter* is a parameter name, a digit, or any of the characters `?`, `$`, or `!`.

A *simple-command* is a sequence of words separated by blanks. The first word specifies the pathname or filename of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument `0` [see exec(BA_OS)]. The *value* of a simple-command is its exit status if it terminates normally, or (octal) `200+`*status* if it terminates abnormally [see wait(BA_OS)].

A *pipeline* is a sequence of two or more simple-commands separated by the character `|`. The standard output of each simple-command (except the last simple-command in the sequence) is connected by a `pipe()` routine to the standard input of the next simple-command. Each simple-command is run as a separate process; the command execution process waits for the last simple-command to terminate. The exit status of a pipeline is the exit status of the last command.

A *command* is either a simple-command or a *list* enclosed in parentheses. Unless otherwise stated, the value returned by a command is that of the last simple-command executed in the command.

A *list* is a command, a pipeline or a sequence of commands and pipelines separated by the character `;` or `&` or the character-pair `&&` or `||`. Of these, the characters `;` and `&`

Comments

A word beginning with the character # causes that word and all the following characters up to a newline to be ignored.

Command Substitution

The standard output from a command enclosed within grave-accents (‘ ‘) may be used as part or all of a word; trailing newlines are removed.

Parameter Substitution

The character $ is used to introduce substitutable keyword-parameters.

${*parameter*} The value, if any, of the *parameter* is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name.

Keyword-parameters (also known as variables) may be assigned values by writing:

parameter-name = value

The following parameters are automatically set:

Parameter	*Description*
?	The decimal value returned by the last synchronously executed command in this call to `system()`.
$	The process-number of this process.
!	The process-number of the last background command invoked in this call to `system()`.

The following parameters are used by the command execution process:

Parameter	*Description*
HOME	The initial working (home) directory.
PATH	The search path for commands (see **Execution**, below).

Blank Interpretation

After parameter and command substitution, the results of substitution are scanned for internal field separator characters (*space*, *tab* and *newline*) and split into distinct arguments where such characters are found. Explicit null arguments ("" or ‘’) are retained. Implicit null arguments (those resulting from parameters that have no values) are removed.

File Name Generation

Following substitution, each word in the command is scanned for the characters *, ?, and [. If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no filename is found that matches the pattern, the word is left unchanged. The character . at the start of a filename or immediately following the character /, as well as the character / itself, must be matched explicitly.

Parameter	*Description*
*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by the character – matches any character lexically between the pair, inclusive. If the first character following the opening [is the character ! any character *not* enclosed is matched.

Quoting

The following characters have special meaning and cause termination of a word unless enclosed in quotation marks as explained below:

; & ( ) | < > *newline space tab*

A character may be *quoted* (*i.e.*, made to stand for itself) by preceding it with the character \. The character-pair *newline* is ignored. All characters enclosed between a pair of single quote marks (′ ′), except a single quote, are quoted. Inside double quote marks (" "), parameter and command substitution occurs and the character \ quotes the characters \ , * , " , and $.

Input/Output

Before a command is executed, its input and output may be redirected using a special notation. The following may appear anywhere in a simple-command, or may precede or follow a command, and are not passed on to the invoked command; substitution occurs before word or digit is used:

Notation	*Description*
<*word*	Use file *word* as standard input (file descriptor 0).
>*word*	Use file *word* as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.
>>*word*	Use file *word* as standard output. If the file exists, output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
<&*digit*	Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using >&*digit*.
<&-	The standard input is closed. Similarly for the standard output using >&-.

If a digit precedes any of the above, the digit specifies the file descriptor to be associated with the file (instead of the default 0 or 1). For example:

... 2>&1

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. Redirections are

first associates file descriptor 1 with file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (*i.e.*, *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

If a command is followed by the character & the default standard input for the command is the empty file /dev/null. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking process as modified by input/output specifications.

Environment

The *environment* [see exec(BA_OS)] is a list of parameter name-value pairs passed to an executed program in the same way as a normal argument list. On invocation, the environment is scanned and a parameter is created for each name found, giving it the corresponding value.

The environment for any simple-command may be augmented by prefixing it with one or more assignments to parameters. For example:

```
TERM=450 cmd ;
```

Signals

The SIGINT and SIGQUIT signals for an invoked command are ignored if the command is followed by the character &; otherwise signals have the values inherited by the command execution process from its parent.

Execution

The above substitutions are carried out each time a command is executed. A new process is created and an attempt is made to execute the command via the exec routines [see exec(BA_OS)].

The parameter PATH defines the search path for the directory containing the command. The character : separates pathnames. NOTE: The current directory is specified by a null pathname, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If the command name contains the character / the search path is not used. Otherwise, each directory in the path is searched for an executable file until the first such executable is found or until the last directory in the path is searched.

RETURN VALUE

If the argument is a null pointer, system() returns non-zero only if a command processor is available. If the argument is not a null pointer, and upon successful completion, the function system() returns the exit status of the command language interpreter in the format specified by waitpid()[see wait(BA_OS)]. Errors, such as syntax errors, cause a non-zero return value and execution of the command is abandoned. Otherwise the function system() returns a value of -1 and sets errno to indicate the error.

ERRORS

Under the following conditions, the function system() fails and sets errno to:

EAGAIN if the system imposed limit on the total number of processes under execution system wide {PROC_MAX} or by a single user ID {CHILD_MAX} would be exceeded.

`EINTR` if the function `system()` was interrupted by a signal.

`ENOMEM` if the process requires more space than the system is able to supply.

FILES

`/dev/null`

USAGE

If possible, applications should use the the function `system()`, which is easier to use and supplies more functions, rather than the `fork()` and exec routines.

SEE ALSO

dup(BA_OS), exec(BA_OS), fork(BA_OS), passwd(BA_ENV), pipe(BA_OS), signal(BA_ENV), ulimit(BA_OS), umask(BA_OS), wait(BA_OS).

LEVEL

Level 1.

NAME

telldir – current location of a named directory stream

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

long telldir(DIR *dirp);
```

DESCRIPTION

The function `telldir()` returns the current location associated with the named directory.

RETURN VALUE

Upon successful completion, the function `telldir()` returns the current location.

SEE ALSO

directory(BA_OS), seekdir(BA_OS).

LEVEL

Level 1.

NAME

termios: tcgetattr, tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow, cfgetospeed, cfgetispeed, cfsetispeed, cfsetospeed, tcgetpgrp, tcsetpgrp, tcgetsid – get and set terminal attributes, line control, get and set baud rate, get and set terminal foreground process group ID, get terminal session ID

SYNOPSIS

```
#include <termios.h>
#include <unistd.h>

int tcgetattr(int fildes, struct termios *termios_p);

int tcsetattr(int fildes, int optional_actions, struct termios *termios_p);

int tcsendbreak(int fildes, int duration);

int tcdrain(int fildes);

int tcflush(int fildes, int queue_selector);

int tcflow(int fildes, int action);

speed_t cfgetospeed(struct termios *termios_p);

int cfsetospeed(struct termios *termios_p, speed_t speed);

speed_t cfgetispeed(struct termios *termios_p);

int cfsetispeed(struct termios *termios_p, speed_t speed);

#include <sys/types.h>
#include <termios.h>

pid_t tcgetpgrp(int fildes);

int tcsetpgrp(int fildes, pid_t pgid);

pid_t tcgetsid(int fildes);
```

DESCRIPTION

The termios functions describe a general terminal interface that is provided to control asynchronous communications ports. A more detailed overview of the terminal interface can be found in termio(BA_DEV). That section also describes an `ioctl()` interface that can be used to access the same functionality. However, the function interface described here is the preferred user interface.

Many of the functions described here have a *termios_p* argument that is a pointer to a `termios` structure. This structure contains the following members:

```
tcflag_t  c_iflag;          /* input modes */
tcflag_t  c_oflag;          /* output modes */
tcflag_t  c_cflag;          /* control modes */
tcflag_t  c_lflag;          /* local modes */
cc_t      c_cc[NCCS];       /* control chars */
```

These structure members are described in detail in termio(BA_DEV).

The `tcgetattr()` function gets the parameters associated with the object referred by *fildes* and stores them in the `termios` structure referenced by *termios_p*. This function may be invoked from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

The `tcsetattr()` function sets the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the `termios` structure referenced by *termios_p* as follows:

If *optional_actions* is `TCSANOW`, the change occurs immediately.

If *optional_actions* is `TCSADRAIN`, the change occurs after all output written to *fildes* has been transmitted. This function should be used when changing parameters that affect output.

If *optional_actions* is `TCSAFLUSH`, the change occurs after all output written to the object referred by *fildes* has been transmitted, and all input that has been received but not read will be discarded before the change is made.

The symbolic constants for the values of *optional_actions* are defined in `<termios.h>`.

If the terminal is using asynchronous serial data transmission, the `tcsendbreak()` function causes transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, it causes transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5 seconds. If *duration* is not zero, zero-valued bits are not transmitted.

If the terminal is not using asynchronous serial data transmission, the `tcsendbreak()` function sends data to generate a break condition or returns without taking any action.

The `tcdrain()` function waits until all output written to the object referred to by *fildes* has been transmitted.

The `tcflush()` function discards data written to the object referred to by *fildes* but not transmitted, or data received but not read, depending on the value of *queue_selector*:

If *queue_selector* is `TCIFLUSH`, it flushes data received but not read.

If *queue_selector* is `TCOFLUSH`, it flushes data written but not transmitted.

If *queue_selector* is `TCIOFLUSH`, it flushes both data received but not read, and data written but not transmitted.

The `tcflow()` function suspends transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*:

If *action* is `TCOOFF`, it suspends output.

If *action* is `TCOON`, it restarts suspended output.

If *action* if `TCIOFF`, the system transmits a STOP character, which is intended to cause the terminal device to stop transmitting data to the system.

If *action* is `TCION`, the system transmits a START character, which is intended to cause the terminal device to start transmitting data to the system.

The baud rate functions are provided for getting and setting the values of the input and output baud rates in the `termios` structure. The effects on the terminal device described below do not become effective until the `tcsetattr()` function is successfully called.

The input and output baud rates are stored in the `termios` structure. The values shown in the table are supported. The names in this table are defined in <`termios.h`>.

Name	Description	Name	Description
`B0`	Hang up	`B600`	600 baud
`B50`	50 baud	`B1200`	1200 baud
`B75`	75 baud	`B1800`	1800 baud
`B110`	110 baud	`B2400`	2400 baud
`B134`	134.5 baud	`B4800`	4800 baud
`B150`	150 baud	`B9600`	9600 baud
`B200`	200 baud	`B19200`	19200 baud
`B300`	300 baud	`B38400`	38400 baud

`cfgetospeed()` gets the output baud rate and stores it in the `termios` structure pointed to by *termios_p*.

`cfsetospeed()` sets the output baud rate stored in the `termios` structure pointed to by *termios_p* to *speed*. The zero baud rate, `B0`, is used to terminate the connection. If `B0` is specified, the modem control lines are no longer asserted. Normally, this will disconnect the line.

`cfgetispeed()` returns the input baud rate stored in the `termios` structure pointed to by *termios_p*.

`cfsetispeed()` sets the input baud rate stored in the `termios` structure pointed to by *termios_p* to *speed*. If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate. Attempts to set unsupported baud rates will be ignored. This refers both to changes to baud rates not supported by the hardware, and to changes setting the input and output baud rates to different values if the hardware does not support this.

`tcsetpgrp()` sets the foreground process group ID of the terminal specified by *fildes* to *pgid*. The file associated with *fildes* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgid* must match a process group ID of a process in the same session as the calling process.

`tcgetpgrp()` returns the foreground process group ID of the terminal specified by *fildes*. The function `tcgetpgrp()` is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

`tcgetsid()` returns the session ID of the terminal specified by *fildes*.

RETURN VALUE

Upon successful completion, the function `tcgetpgrp()` returns the process group ID of the foreground process group associated with the terminal; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

Upon successful completion, `tcgetsid()` returns the session ID associated with the terminal. Otherwise, a value of `-1` is returned and `errno` is set to indicate an error.

Upon successful completion, `cfgetispeed()` returns the input baud rate stored in the `termios` structure.

Upon successful completion, `cfgetospeed()` returns the output baud rate stored in the `termios` structure.

Upon successful completion, all other functions return a value of `0`. Otherwise, a value of `-1` is returned and `errno` is set to indicate an error.

ERRORS

Under the following conditions, the described functions fail and set `errno` to:

`EBADF` if the *fildes* argument is not a valid file descriptor.

`ENOTTY` if the file associated with *fildes* is not a terminal.

Additionally, specific functions fail and set `errno` as follows:

Under the following conditions, the function `tcsetattr()` fails and sets `errno` to:

`EINVAL` if the *optional_actions* argument is not a proper value, or an attempt was made to change an attribute represented in the `termios` structure to an unsupported value.

Under the following conditions, the function `tcsendbreak()` fails and sets `errno` to:

`EINVAL` if the device does not support the `tcsendbreak()` function.

Under the following conditions, the function `tcdrain()` fails and sets `errno` to:

`EINTR` if a signal interrupted the `tcdrain()` function.

`EINVAL` if the device does not support the `tcdrain()` function.

Under the following conditions, the function `tcflush()` fails and sets `errno` to:

`EINVAL` if the device does not support the `tcflush()` function, or the *queue_selector* argument is not a proper value.

Under the following conditions, the function `tcflow()` fails and sets `errno` to:

`EINVAL` if the device does not support the `tcflow()` function or the *action* argument is not a proper value.

Under the following conditions, the function `tcgetpgrp()` fails and sets `errno` to:

`ENOTTY` if the calling process does not have a controlling terminal, or the file is not the controlling terminal.

Under the following conditions, the function `tcsetpgrp()` fails and sets `errno` to:

`EPERM` if *pgid* does not match the process group of an existing process in the same session as the calling process.

`EINVAL` if the value of the *pgid* argument is not a valid process group ID.

`ENOTTY` if the calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

Under the following conditions, the function `tcgetsid()` fails and sets `errno` to:

`EACCES` if *fildes* is a terminal that is not allocated to a session.

SEE ALSO

setsid(BA_OS), setpgid(BA_OS), termios(BA_ENV).

LEVEL

Level 1.

NAME

time – get time

SYNOPSIS

```
#include <sys/types.h>
#include <time.h>

time_t time(time_t *tloc);
```

DESCRIPTION

The function `time()` returns the value of time in seconds since 00:00:00 UTC, January 1, 1970.

As long as the argument *tloc* is not a null pointer, the return value is also stored in the location to which the argument *tloc* points.

The actions of the function `time()` are undefined if the argument *tloc* points to an invalid address.

RETURN VALUE

Upon successful completion, the function `time()` returns the value of time; otherwise, it returns `(time_t)-1`.

SEE ALSO

stime(BA_OS).

LEVEL

Level 1.

NAME

times – get process and child process times

SYNOPSIS

```
#include <sys/types.h>
#include <sys/times.h>

clock_t times(struct tms *buffer);
```

DESCRIPTION

times fills the **tms** structure pointed to by *buffer* with time-accounting information. The **tms** structure is defined in **sys/times.h** and includes the following fields:

```
clock_t    tms_utime;
clock_t    tms_stime;
clock_t    tms_cutime;
clock_t    tms_cstime;
```

This information comes from the calling process and each of its terminated child processes for which it has executed a wait routine. All times are reported in clock ticks. The clock ticks at a system-dependent rate. The specific value of this rate for an implementation is defined, in ticks per second, by the variable **CLK_TCK**, found in the include file **limits.h**.

tms_utime is the **SM** time used while executing instructions in the user space of the calling process.

tms_stime is the **SM** time used by the system on behalf of the calling process.

tms_cutime is the sum of the **tms_utime** and the **tms_cutime** of the child processes.

tms_cstime is the sum of the **tms_stime** and the **tms_cstime** of the child processes.

Return Values

On success, **times** returns the elapsed real time in clock ticks from an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of **times** to another. On failure, **times** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **times** fails and sets **errno** to:

EFAULT *buffer* points to an invalid address.

SEE ALSO

exec(BA_OS), **fork**(BA_OS), **wait**(BA_OS), **waitid**(BA_OS), **waitpid**(BA_OS),

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

Statistics are gathered at the process level and represent the combined usage of all contained threads.

NAME

ulimit – get and set user limits

SYNOPSIS

```
#include <ulimit.h>

long ulimit(int cmd, ... /* arg */);
```

DESCRIPTION

The function `ulimit()` provides for control over process limits.

Values available for the argument *cmd* are:

`UL_GETFSIZE`
: Get the file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

`UL_SETFSIZE`
: Set the file size limit of the process equal to *arg*, taken as a `long`. Any process may decrease this limit, but only a process with appropriate privileges may increase the limit. The new file size limit is returned.

RETURN VALUE

Upon successful completion, the function `ulimit()` returns a non-negative value; otherwise, it returns a value of `-1`, the limit is unchanged and `errno` is set to indicate an error.

ERRORS

Under the following condition, the function `ulimit()` fails and sets `errno` to:

`EINVAL` if the *cmd* argument is not valid.

`EPERM` if a process not having appropriate privileges attempts to increase its file size limit.

SEE ALSO

getrlimit(BA_OS), write(BA_OS).

FUTURE DIRECTIONS

To be removed in a future issue of the SVID.

LEVEL

Level 1.

NAME

ustat – get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat(dev_t dev, struct ustat *buf);
```

DESCRIPTION

ustat returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system [see **makedev**(3C)]. *buf* is a pointer to a **ustat** structure that includes the following elements:

```
daddr_t  f_tfree;     /* Total free blocks */
ino_t    f_tinode;    /* Number of free inodes */
char     f_fname[6];  /* Filsys name */
char     f_fpack[6];  /* Filsys pack name */
```

Return Values

On success, **ustat** returns 0. On failure, **ustat** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **ustat** fails and sets **errno** to:

EINVAL *dev* is not the device number of a device containing a mounted file system.

EFAULT *buf* points outside the process's allocated address space.

EINTR A signal was caught during a **ustat** system call.

ENOLINK *dev* is on a remote machine and the link to that machine is no longer active.

ECOMM *dev* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

stat(BA_OS)

LEVEL

Level 2.

NAME

umask – set and get file creation mask

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

DESCRIPTION

umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the access permission bits of *cmask* and the file mode creation mask are used.

Return Values

umask returns the previous value of the file mode creation mask.

SEE ALSO

chmod (BA_OS), **creat** (BA_OS), **mkdir** (BA_OS), **mknod** (BA_OS), **open** (BA_OS), **sh** (BU_CMD), **stat** (BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

The file creation mask is an attribute of the containing process and is shared by sibling threads.

NAME

umount – unmount a file system

SYNOPSIS

```
#include <sys/mount.h>

int umount(const char *file);
```

DESCRIPTION

umount requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *file* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

umount may be invoked only by a process with appropriate privileges.

Return Values

On success, **umount** returns 0. On failure, **umount** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **umount** fails and sets **errno** to:

EBUSY	A file on *file* is busy.
EINVAL	*file* does not exist.
EINVAL	*file* is not mounted.
ELOOP	Too many symbolic links were encountered in translating the path pointed to by *file*.
ENAMETOOLONG	The length of the *file* argument exceeds {**PATH_MAX**}, or the length of a *file* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.
ENOTDIR	*file* does not point to a directory.
ENOENT	A component of the path prefix does not exist or is a null pathname.
ENOTBLK	*file* is not a block special device.
EPERM	The calling process does not have the appropriate privilege.

NOTICES

umount will now resolve the **mount_point** argument using **realpath**(3C) before any processing is performed.

USAGE

The function **umount** is not recommended for use by application programs.

SEE ALSO

mount(BA_OS).

LEVEL

Level 1.

NAME

uname – get name of current operating system

SYNOPSIS

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

DESCRIPTION

The function `uname()` stores information identifying the current operating system in the structure pointed to by the argument *name.*

The function `uname()` uses the `utsname` structure defined by the `<sys/utsname.h>` header file whose members include:

```
char  sysname[{SYS_NMLN}];
char  nodename[{SYS_NMLN}];
char  release[{SYS_NMLN}];
char  version[{SYS_NMLN}];
char  machine[{SYS_NMLN}];
```

The function `uname()` returns a null-terminated character string naming the current operating system in the character array `sysname`.

Similarly, the character array `nodename` contains the name that the system is known by on a communications network.

The members `release` and `version` further identify the operating system.

The member `machine` contains a standard name that identifies the hardware on which the operating system is running.

RETURN VALUE

Upon successful completion, the function `uname()` returns a non-negative value; otherwise, it returns a value of `–1` and sets `errno` to indicate an error.

LEVEL

Level 1.

NAME

unlink – remove directory entry

SYNOPSIS

```
#include <unistd.h>

int unlink(const char *path);
```

DESCRIPTION

The function `unlink()` removes the directory entry named by the pathname pointed to by the argument *path* and decrements the link count of the file referenced by the directory entry. When all links to a file have been removed and no process has an outstanding reference to the file, the space occupied by the file is freed and the file ceases to exist. If one or more processes have outstanding references to the file when the last link is removed, space occupied by the file is not released until all references to the file have been removed. If *path* is a symbolic link, the symbolic link is removed. The *path* argument should not name a directory unless the process has appropriate privileges and the implementation supports `unlink()` on directories. Applications should use `rmdir()` to remove directories.

Upon successful completion the function `unlink()` marks for update the `st_ctime` and `st_mtime` fields of the parent directory. Also, if the file's link count is not zero, the `st_ctime` field of the file is marked for update.

RETURN VALUE

Upon successful completion, the function `unlink()` returns `0`; otherwise, it returns `-1`, the named file is not changed and `errno` is set to indicate an error.

ERRORS

Under the following conditions, the function `unlink()` fails and sets `errno` to:

`ENOTDIR`	if a component of the path prefix is not a directory.
`ENOENT`	if the named file does not exist, or *path* points to an empty string.
`EACCES`	if a component of the path prefix denies search permission.
`EACCES`	if the directory containing the link to be removed denies write permission.
`EPERM`	if the named file is a directory and the process does not have appropriate privileges.
`EBUSY`	if the entry to be unlinked is the mount point for a mounted file system.
`EROFS`	if the directory entry to be unlinked is part of a read-only file system.
`ENAMETOOLONG`	if the length of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
`ELOOP`	if too many symbolic links are encountered in translating the path.

SEE ALSO

close(BA_OS), open(BA_OS), remove(BA_OS), rmdir(BA_OS) unlink(BA_OS).

LEVEL

Level 1.

NAME

utime – set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
#include <utime.h>

int utime(const char *path, const struct utimbuf *times);
```

DESCRIPTION

The function `utime()` sets the access and modification times of the named file.

The argument *path* points to a pathname naming a file.

If the argument *times* is null, the access and modification times of the file are set to the current time. A process must be the owner of the file or have appropriate privileges to use the function `utime()` in this manner.

If the argument *times* is not null, *times* is interpreted as a pointer to a structure `utimbuf` (see below), and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or a process with appropriate privileges may use the function `utime()` this way.

The times in the structure `utimbuf` are measured in seconds since 00:00:00 UTC Jan. 1, 1970.

The structure `utimbuf` contains the following members:

```
time_t actime;  /* access time */
time_t modtime; /* modification time */
```

The function `utime()` also causes the time of the last file status change (`st_ctime`) to be updated [see stat(BA_OS)].

RETURN VALUE

Upon successful completion, the function `utime()` returns a value of `0`; otherwise, it returns a value of `-1`, the file times are not affected and `errno` is set to indicate an error.

ERRORS

Under the following conditions, the function `utime()` fails and sets `errno` to:

`ENOENT`	if the named file does not exist, or *path* points to an empty string.
`ENOTDIR`	if a component of the path prefix is not a directory.
`EACCES`	if a component of the path prefix denies search permission.
`EPERM`	if the effective user ID does not match the owner of the file or does not have the appropriate privileges and the argument *times* is not null.
`EACCES`	if the effective user ID does not match the owner of the file, or does not have the appropriate privileges and the argument *times* is null and write access is denied.
`EROFS`	if the file system containing the file is mounted read-only.

ENAMETOOLONG
: if the length of a pathname exceeds {PATH_MAX}, or pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

ELOOP
: if too many symbolic links are encountered in translating the path.

SEE ALSO

stat(BA_OS), utime(BA_ENV).

LEVEL

Level 1.

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

DESCRIPTION

wait suspends the calling process until one of its immediate children terminates or until a child that is being traced stops because it has received a signal. The **wait** system call will return prematurely if a signal is received. If all child processes stopped or terminated prior to the call on **wait**, return is immediate.

If **wait** returns because the status of a child process is available, it returns the process ID of the child process. If the calling process had specified a non-zero value for *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*. It may be evaluated with the macros described on **wstat**. In the following, *status* is the object pointed to by *stat_loc*:

If the child process stopped, the high order 8 bits of *status* will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to **WSTOPFLG**.

If the child process terminated due to an **exit** call, the low order 8 bits of *status* will be 0 and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to **exit**. [see **exit**(BA_OS)].

If the child process terminated due to a signal, the high order 8 bits of *status* will be 0 and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if **WCOREFLG** is set, a ''core image'' will have been produced. [see **signal**(BA_OS)].

If **wait** returns because the status of a child process is available, then that status may be evaluated with the macros defined by **wstat**.

If a parent process terminates without waiting for its child processes to terminate, the parent process **ID** of each child process is set to 1. This means the initialization process inherits the child processes.

Return Values

If **wait** returns due to a stopped or terminated child process, the process **ID** of the child is returned to the calling process. Otherwise, **wait** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **wait** fails and sets **errno** to:

ECHILD The calling process has no existing unwaited-for child processes.

EINTR The function was interrupted by a signal.

SEE ALSO

exec(BA_OS), **fork**(BA_OS), **pause**(BA_OS), **ptrace**(KE_OS), **signal**(BA_OS),

LEVEL

Level 1.

NOTICES

See NOTICES in **signal**(BA_OS).

If **SIGCLD** is held, then **wait** does not recognize death of children.

Considerations for Threads Programming

While one thread is blocked, siblings might still be executing.

NAME

waitid – wait for child process to change state

SYNOPSIS

```
#include <sys/types.h>
#include <wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop,
      int options);
```

DESCRIPTION

waitid suspends the calling process until one of its children changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to **waitid**, **waitid** returns immediately.

The *idtype* and *id* arguments specify which children **waitid** is to wait for.

If *idtype* is **P_PID**, **waitid** waits for the child with a process ID equal to **(pid_t)** *id*.

If *idtype* is **P_PGID**, **waitid** waits for any child with a process group ID equal to **(pid_t)** *id*.

If *idtype* is **P_ALL**, **waitid** waits for any children and *id* is ignored.

The *options* argument is used to specify which state changes *waitid* is to wait for. It is formed by an OR of any of the following flags:

WEXITED	Wait for process(es) to exit.
WTRAPPED	Wait for traced process(es) to become trapped or reach a breakpoint [see **ptrace**(KE_OS)].
WSTOPPED	Wait for and return the process status of any child that has stopped upon receipt of a signal.
WCONTINUED	Return the status for any child that was stopped and has been continued.
WNOHANG	Return immediately.
WNOWAIT	Keep the process in a waitable state. This will not affect the state of the process on subsequent waits.

infop must point to a **siginfo_t** structure, as defined in **siginfo**. **siginfo_t** is filled in by the system with the status of the process being waited for.

Return Values

If **waitid** returns due to a change of state of one of its children, it returns 0. Otherwise, **waitid** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **waitid** fails and sets **errno** to:

EFAULT	*infop* points to an invalid address.
EINTR	**waitid** was interrupted due to the receipt of a signal by the calling process.

EINVAL	0 or another invalid value was specified for *options*.
EINVAL	*idtype* and *id* specify an invalid set of processes.
ECHILD	The set of processes specified by *idtype* and *id* does not contain any unwaited-for processes.

SEE ALSO

exec(2), **exit**(2), **fork**(2),

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

While one thread is blocked, siblings might still be executing.

NAME

waitpid – wait for child process to change state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid (pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

waitpid suspends the calling process until one of its children changes state; if a child process changed state prior to the call to **waitpid**, return is immediate. *pid* specifies a set of child processes for which status is requested.

If *pid* is equal to **(pid_t)–1**, status is requested for any child process.

If *pid* is greater than **(pid_t)0**, it specifies the process ID of the child process for which status is requested.

If *pid* is equal to **(pid_t)0** status is requested for any child process whose process group ID is equal to that of the calling process.

If *pid* is less than **(pid_t)–1**, status is requested for any child process whose process group ID is equal to the absolute value of *pid* .

If **waitpid** returns because the status of a child process is available, then that status may be evaluated with the macros defined by If the calling process had specified a non-zero value of *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*.

The *options* argument is constructed from the bitwise inclusive **OR** of zero or more of the following flags, defined in the header file **sys/wait.h**:

WCONTINUED the status of any continued child process specified by *pid*, whose status has not been reported since it continued (from a job control stop), shall also be reported to the calling process.

WNOHANG **waitpid** will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid* .

WNOWAIT keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with identical results.

WUNTRACED the status of any child processes specified by *pid*^ that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the calling process.

waitpid with *options* equal to **WUNTRACED** and *pid*^ equal to (**pid_t**)–1 is identical to a call to **wait**(BA_OS).

Return Values

If **waitpid** returns because the status of a child process is available, it returns the process ID of the child process for which status is reported. If **waitpid** was invoked with **WNOHANG** set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, **waitpid** returns 0. Otherwise, **waitpid** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **waitpid** fails and sets **errno** to:

EINTR	**waitpid** was interrupted due to the receipt of a signal sent by the calling process.
EINVAL	An invalid value was specified for *options.*
ECHILD	The process or process group specified by *pid* does not exist or is not a child of the calling process or can never be in the states specified by *options*.

SEE ALSO

exec(BA_OS), **exit**(BA_OS), **fork**(BA_OS), **pause**(BA_OS), **ptrace**(KE_OS), **sigaction**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

While one thread is blocked, siblings might still be executing.

NAME

write, **writev** – write on a file

SYNOPSIS

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);

#include <sys/types.h>
#include <sys/uio.h>

ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with *fildes*. If *nbyte* is 0 and the file is a regular file, **write** returns 0 and has no other results. If the value of *nbyte* is greater than **{SSIZE_MAX}** the result is undefined. *fildes* is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, **pipe**, or **ioctl** system call.

writev performs the same action as **write**, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt* – 1]. The *iovcnt* is valid only if greater than 0 and less than or equal to **{IOV_MAX}**.

For **writev**, the **iovec** structure contains the following members:

```
void *   iov_base;
int       iov_len;
```

Each **iovec** entry specifies the base address and length of an area in memory from

If **O_NONBLOCK** is set, **write** returns **-1** and sets **errno** to **EAGAIN**.

If **O_NONBLOCK** is clear, **write** sleeps until all blocking locks are removed or the **write** is terminated by a signal.

If a **write** requests that more bytes be written than there is room for—for example, if the write would exceed the process file size limit [see **getrlimit**(BA_OS) and **ulimit**(BA_OS)], the system file size limit, or the free space on the device—only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A **write** of 512-bytes returns 20. The next **write** of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.

Write requests of **{PIPE_BUF}** bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than **{PIPE_BUF}** bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether the **O_NONBLOCK** flag is are set.

If **O_NONBLOCK** and **O_NDELAY** are clear, a write request may cause the process to block, but on normal completion it returns *nbyte*.

If **O_NONBLOCK** is set, **write** requests are handled in the following way: the **write** does not block the process; write requests for **{PIPE_BUF}** or fewer bytes either succeed completely and return *nbyte*, or return **-1** and set **errno** to **EAGAIN**. A **write** request for greater than **{PIPE_BUF}** bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns **-1** with **errno** set to **EAGAIN**. Also, if a request is greater than **{PIPE_BUF}** bytes and all data previously written to the pipe has been read, **write** transfers at least **{PIPE_BUF}** bytes.

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports nonblocking writes and cannot accept the data immediately:

If **O_NONBLOCK** is clear, **write** blocks until the data can be accepted.

If **O_NONBLOCK** is set, **write** does not block the process. If some data can be written without blocking the process, **write** writes what it can and returns the number of bytes written. Otherwise, if **O_NONBLOCK** is set, it returns **–1** and sets **errno** to **EAGAIN**.

For STREAMS files the operation of **write** is determined by the values of the minimum and maximum *nbyte* range (‘‘packet size’’) accepted by the stream. These values are contained in the topmost stream module. Unless the user pushes the topmost module [see **I_PUSH** in **streams**(BA_DEV)], these values can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes are written. If *nbyte* does not fall within the range and the minimum packet size value is 0, **write** breaks the buffer into maximum packet size segments prior to sending the data downstream (the last segment may be smaller than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero,

write fails and sets **errno** to **ERANGE**. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends a zero-length message with 0 returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and 0 is returned. The user program may issue the **I_SWROPT ioctl**(BA_OS) to enable zero-length messages to be sent across the pipe or FIFO [see **streams**(BA_DEV)].

When writing to a stream, data messages are created with a priority band of 0. When writing to a stream that is not a pipe or FIFO:

If **O_NONBLOCK** is not set, and the stream cannot accept data (the stream write queue is full because of internal flow control conditions), **write** blocks until data can be accepted.

If **O_NONBLOCK** is not set, and the and the stream cannot accept data, **write** returns **-1** and sets **errno** to **EAGAIN**.

If **O_NONBLOCK** is not set, and the part of the buffer has already been written when a condition occurs in which the stream cannot accept additional data, **write** terminates and returns the number of bytes written.

Return Values

On success, **write** and **writev** return the number of bytes actually written and mark for update the **st_ctime** and **st_mtime** fields of the file. On failure, **write** and **writev** return –1 and set **errno** to identify the error.

Errors

In the following conditions, **write** and **writev** fail and set **errno** to:

EAGAIN	Mandatory file/record locking is set, **O_NONBLOCK** is set, and there is a blocking record lock.
EAGAIN	Total amount of system memory available when reading via raw I/O is temporarily insufficient.
EAGAIN	An attempt is made to write to a stream that can not accept data with the or **O_NONBLOCK** flag set.
EBADF	*fildes* is not a valid file descriptor open for writing.
EDEADLK	The **write** was going to go to sleep and cause a deadlock to occur.
EFAULT	*buf* points outside the process's allocated address space.
EFBIG	An attempt is made to write a file that exceeds the process's file size limit or the maximum file size [see **ulimit**(BA_OS)].
EINTR	A signal was caught during the **write** system call.
EINVAL	An attempt is made to write to a stream linked below a multiplexor.
EIO	The process is in the background and is attempting to write to its controlling terminal whose **TOSTOP** flag is set; the process is neither ignoring nor blocking **SIGTTOU** signals, and the process group of the process is orphaned.

EIO *fildes* points to a device special file that is in the closing state.

ENOLINK *fildes* is on a remote machine and the link to that machine is no longer active.

ENOSR An attempt is made to write to a stream with insufficient STREAMS memory resources available in the system.

ENOSPC During a **write** to an ordinary file, there is no free space left on the device.

ENXIO The device associated with the file descriptor is a block-special or character-special file and the file-pointer value is out of range.

EPIPE and **SIGPIPE** signal
An attempt is made to write to a pipe that is not open for reading by any process.

EPIPE An attempt is made to write to a FIFO that is not open for reading by any process.

ERANGE An attempt is made to write to a stream with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.

ENOLCK Enforced record locking was enabled and **{LOCK_MAX}** regions are already locked in the system.

In addition, in the following conditions **writev** fails and sets **errno** to:

EINVAL *iovcnt* was less than or equal to 0, or greater than 16.

EINVAL An **iov_len** value in the *iov* array was negative.

EINVAL The sum of the **iov_len** values in the *iov* array overflowed a 32-bit integer.

A **write** to a STREAMS file can fail if an error message has been received at the stream head. In this case, **errno** is set to the value included in the error message.

After carrier loss, **M_HANGUP** is set, and a subsequent write will return –1 with **errno** set to **EIO**. To write after disconnecting and reconnecting the line, set the **CLOCAL** flag to tell the driver to ignore the state of the line and the driver will not send **M_HANGUP** to the stream head. If **CLOCAL** is not set, and hangup occurs, the application is responsible for re-establishing the connection.

SEE ALSO

creat(BA_OS), **fcntl**(BA_OS), **lseek**(BA_OS), **open**(BA_OS), **pipe**(BA_OS), **pwrite**(BA_OS), **read**(BA_OS), **ulimit**(BA_OS)

LEVEL

Level 1.

The enforcement mode of file and record locking has moved to Level 2 effective September 30, 1989.

NOTICES

Considerations for Threads Programming

Open file descriptors are a process resource and available to any sibling thread; if used concurrently, actions by one thread can interfere with those of a sibling.

While one thread is blocked, siblings might still be executing.

Base OS Library Routines

The following section contains the manual pages for the BA_LIB library routines.

FINAL COPY
June 15, 1995
File: ba_lib.cov
svid

NAME

abs, labs – return integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs(int i);

long labs(long l);
```

DESCRIPTION

The function `abs()` returns the absolute value of its integer operand. The function `labs()` returns the absolute value of its long operand.

USAGE

In two's complement representation, the absolute value of the negative integer with largest magnitude `{INT_MIN}` or `{LONG_MIN}` is undefined. Some implementations may catch this as an error, but others may ignore it.

SEE ALSO

floor(BA_LIB).

LEVEL

Level 1.

NAME

addsev – define additional severities

SYNOPSIS

```
int addsev(int int_val, const char *string);
```

DESCRIPTION

The function `addsev()` defines additional severities for use in subsequent calls to `pfmt()` or `lfmt()`. `addsev()` associates an integer value *int_val* in the range [5-255] with a character *string*. It overwrites any previous string association between *int_val* and *string*.

If *int_val* is ORed with the *flags* passed to subsequent calls `pfmt()` or `lfmt()`, *string* will be used as severity.

Passing a NULL *string* removes the severity.

Add-on severities are only effective within the applications defining them.

RETURN VALUE

`addsev()` returns 0 in case of success, -1 otherwise.

USAGE

Only the standard severities are automatically displayed per the locale in effect at runtime. An application must provide the means for displaying locale-specific versions of add-on severities.

EXAMPLE

```
#define PANIC   5
setlabel("APPL");
setcat("my_appl");
addsev(PANIC, gettxt(":26", "Panic"));
/* ... */
lfmt(stderr, MM_SOFT|MM_APPL|PANIC, ":12:Cannot locate database\n");
```

will display the message to *stderr* and forward to the logging service:

```
APPL: Panic: Cannot locate database
```

SEE ALSO

gettxt(BA_LIB), lfmt(BA_LIB), pfmt(BA_LIB).

FUTURE DIRECTIONS

This interface is to be removed when the three-year waiting period has expired.

LEVEL

Level 2, April 1991.

NAME

assert – verify program assertion

SYNOPSIS

```
#include <assert.h>

void assert(int expression);
```

DESCRIPTION

The `assert()` macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), `assert()` prints:

`assertion failed:` *expression*`, file` *xyz*`, line` *nnn*

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the `assert()` statement, the latter are respectively the values of the preprocessor macros `__FILE__` and `__LINE__`.

USAGE

Compiling with the preprocessor option `-DNDEBUG` or with the preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement will stop assertions from being compiled into the program.

SEE ALSO

abort(BA_OS), assert(BA_ENV).

LEVEL

Level 1.

NAME

Bessel: j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0(double x);

double j1(double x);

double jn(int n, double x);

double y0(double x);

double y1(double x);

double yn(int n, double x);
```

DESCRIPTION

The functions `j0()` and `j1()` return Bessel functions of *x* of the first kind of orders 0 and 1, respectively. The function `jn()` returns the Bessel function of *x* of the first kind of order *n*.

The functions `y0()` and `y1()` return Bessel functions of *x* of the second kind of orders 0 and 1, respectively. The function `yn()` returns the Bessel function of *x* of the second kind of order *n*.

For the functions `y0()`, `y1()`, and `yn()`, the argument *x* must be positive.

RETURN VALUE

A macro `HUGE_VAL` will be defined by the `<math.h>` header file. This macro expands to a positive double expression, not necessarily representable as a float. On implementations that support the IEEE 754 standard, `HUGE_VAL` evalutates to $+\infty$.

If an input parameter is `NaN`, then the function will return `NaN` and set errno to `EDOM`.

The functions `y0()`, `y1()`, and `yn()` will return –`HUGE_VAL` when *x* is zero, and set errno to `EDOM`.

The functions `y0()`, `y1()`, and `yn()`, when *x* is negative, will return IEEE `NaN` (Not a Number) if available, or –`HUGE_VAL` otherwise. Errno will be set to `EDOM`.

Values of *x* too large in magnitude cause the functions `j0()`, `j1()`, `jn()`, `y0()`, `y1()`, and `yn()` to return zero and to set `errno` to `ERANGE`.

LEVEL

Level 1

NAME

bsearch – binary search on a sorted table

SYNOPSIS

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base,
     size_t nel, size_t width,
     int (*compar)(const void *, const void *));
```

DESCRIPTION

The function `bsearch()` is a binary search routine. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a user-provided comparison function, *compar()* [see qsort(BA_LIB)].

The argument *key* points to an object to be sought in the table.

The argument *base* points to the element at the base of the table.

The argument *nel* is the number of elements in the table.

The argument *width* is the size of an element in bytes.

The argument *compar* is the name of the comparison function, which is called with two arguments of type `const void *` that point to the elements being compared. The *compar()* function must return an integer less than, equal to or greater than zero, as the first argument is to be considered less than, equal to or greater than the second.

RETURN VALUE

Upon succesful completion, the function `bsearch()` returns a pointer to a matching member of the table. A null pointer is returned if the key cannot be found in the table. If two members compare as equal, the member that is matched is unspecified.

USAGE

The pointers to the key and the element at the base of the table, *key* and *base*, respectively, should be of type pointer-to-element and cast to type `(const void *)`, respectively.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type `void *`, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following example searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry. This code fragment reads in strings; it either finds the corresponding node and prints out the string and its length or it prints an error message.

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

#define TABSIZE 1000
```

```
struct node {       /* these are in the table */
  char *string;
  int length;
};
struct node table[TABSIZE]; /* table to be searched */
    . . .
{
  struct node *node_ptr, node;
  int node_compare();  /* routine to compare 2 nodes */
  char str_space[20];  /* space to read string into */
      . . .
  node.string = str_space;
  while (scanf("%s", node.string) != EOF) {
    node_ptr = (struct node *)bsearch((const void *)(&node),
      (const void *)table, TABSIZE,
      sizeof(struct node), node_compare);
    if (node_ptr != (char*)NULL)
         (void)printf("string = %20s, length = %d\n",
             node_ptr->string, node_ptr->length);
    else
        (void) printf("not found: %s\n", node.string);
  }/* while */
}
/*
    This routine compares two nodes based on an
    alphabetical ordering of the string field.
*/
int node_compare(struct node *node1, struct node *node2);
{
    return strcmp(node1->string, node2->string);
}
.ft 1
```

SEE ALSO

hsearch(BA_LIB), lsearch(BA_LIB), qsort(BA_LIB), tsearch(BA_LIB).

LEVEL

Level 1.

NAME

catgets – read a program message

SYNOPSIS

```
#include <nl_types.h>

char *catgets(nl_catd catd, int set_num, int msg_num, const char
*s);
```

DESCRIPTION

The *catgets* function attempts to read message *msg_num*, in set *set_num*, from the message catalogue identified by *catd*. *catd* is a catalogue descriptor returned from an earlier call to `catopen()` [see catopen(BA_LIB)]. *s* points to a default message string which will be returned by `catgets()` if the identified message catalogue is not currently available.

RETURN VALUE

If the identified message is retrieved successfully, `catgets()` returns a pointer to an internal buffer area containing the null terminated message string. If the call is unsuccessful because the message catalogue identified by *catd* is not currently available, a pointer to *s* is returned.

SEE ALSO

catopen(BA_LIB).

LEVEL

Level 1.

NAME

catopen, **catclose** – open / close a message catalog

SYNOPSIS

```
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);

int catclose(nl_catd catd);
```

DESCRIPTION

catopen opens a message catalog and returns a catalog descriptor. *name* specifies the name of the message catalog to be opened. If *name* contains a ''/'' then *name* specifies a pathname for the message catalog. Otherwise, the environment variable **NLSPATH** is used. If **NLSPATH** does not exist in the environment, or if a message catalog cannot be opened in any of the paths specified by **NLSPATH**, then the default path is used [see nl_types(BA_ENV)].

The names of message catalogs, and their location in the filestore, can vary from one system to another. Individual applications can choose to name or locate message catalogs according to their own special needs. A mechanism is therefore required to specify where the catalog resides.

The **NLSPATH** variable provides both the location of message catalogs, in the form of a search path, and the naming conventions associated with message catalog files. For example:

```
NLSPATH=/nlslib/%L/%N.cat:/nlslib/%N/%L
```

The metacharacter **%** introduces a substitution field, where **%L** substitutes the current setting of the *locale* (see below) and **%N** substitutes the value of the *name* parameter passed to **catopen**. Thus, in the above example, **catopen** will search in **/nlslib/***locale*/*name***.cat**, then in **/nlslib/***name***/***locale*, for the required message catalog.

The evaluation of *locale* as referenced by the substitution field **%L** depends on the argument *oflag*. When *oflag* is **NL_CAT_LOCALE**, the **LC_MESSAGES** category as returned by **setlocale**(BA_OS) is used to locate the message catalog. When *oflag* is zero, the environment variable **LANG** locates the catalog without regard to the **LC_MESSAGES** category. If either of these methods fails, then the default language as defined in **nl_types.h** is used.

For a complete description of the metacharacters available for **NLSPATH**, see **envvar**(BA_ENV).

NLSPATH will normally be set up on a system wide basis (for example, in **/etc/profile**) and thus makes the location and naming conventions associated with message catalogs transparent to both programs and users.

catclose closes the message catalog identified by *catd*.

Return Values

If successful, **catopen** returns a message catalog descriptor for use in subsequent calls to **catgets** and **catclose**. Otherwise **catopen** returns **(nl_catd)-1**.

catclose returns zero if successful, otherwise –1.

SEE ALSO

catgets(BA_LIB), **envvar**(BA_ENV), **nl_types**(BA_ENV), **setlocale**(BA_OS).

LEVEL

Level 1.

NAME

clock – report CPU time used

SYNOPSIS

```
#include <time.h>

clock_t clock(void);
```

DESCRIPTION

The function `clock()` returns the amount of CPU time used since the first call to the function `clock()`. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed the `wait()`, `pclose()`, or `system()` routines.

To determine the time in seconds, the value returned by the `clock()` function should be divided by the value of the macro `CLOCKS_PER_SEC` (the number per second of the value returned by the `clock()` function).

RETURN VALUE

If the processor time used is not available or its value cannot be represented, the function returns the value `(clock_t)-1`.

USAGE

The value returned by `clock()` is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution.

SEE ALSO

times(BA_OS), wait(BA_OS), popen(BA_OS), system(BA_OS).

LEVEL

Level 1.

NAME

conv: toupper, tolower, _toupper, _tolower, toascii – translate characters

SYNOPSIS

```
#include <locale.h>
#include <ctype.h>

int toupper(int c);

int tolower(int c);

int _toupper(int c);

int _tolower(int c);

int toascii(int c);
```

DESCRIPTION

The functions `toupper()` and `tolower()` have as domain the range of the `getc()` routine: an integer, the value of which is representable as an `unsigned char`, or `EOF`, which is defined by the `<stdio.h>` header file and represents end-of-file. If the argument of `toupper()` represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of `tolower()` represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros `_toupper()`, `_tolower()`, and `toascii()` are defined by the `<ctype.h>` header file. The macros `_toupper()` and `_tolower()` accomplish the same thing as `toupper()` and `tolower()`, but have restricted domains and are faster. The macro `_toupper()` requires an lower-case letter as its argument; its result is the corresponding upper-case letter. The macro `_tolower()` requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

The macro `toascii()` yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

The functions `toupper()` and `tolower()` and the macros `_toupper()` and `_tolower()` are affected by `LC_CTYPE`. In the `"C"` locale, or in a locale where shift information is not defined, these functions determine the case of characters according to the rules of the ASCII-coded character set. Characters outside the ASCII range of characters are returned unchanged.

SEE ALSO

ctype(BA_LIB), getc(BA_LIB), setlocale(BA_OS).

LEVEL

Level 1.

NAME

crypt, setkey, encrypt – generate string encoding

SYNOPSIS

```
char *crypt(const char *key, const char *salt);

void setkey(const char *key);

void encrypt(char *block, int edflag);
```

DESCRIPTION

The function `crypt()` is a string-encoding function.

The argument *key* is a string to be encoded. The argument *salt* is a two-character string chosen from the set `[a–zA–Z0–9./]`; this string is used to perturb the encoding algorithm, after which the string that *key* points to is used as the key to repeatedly encode a constant string. The returned value points to the encoded string. The first two characters are the *salt* itself.

The functions `setkey()` and `encrypt()` provide (rather primitive) access to the encoding algorithm. The argument to `setkey()` is a 64-bit string represented by a character array of length 64 containing only the characters with numerical value `0` and `1`. The string is divided into groups of 8 and the low-order bit in each group is ignored; this gives a 56-bit key. This is the key that will be used with the above mentioned algorithm to encode the string *block* with the function `encrypt()`.

The argument to `encrypt()` is a character array of length 64 containing only the characters with numerical value `0` and `1`. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the encoding algorithm using the key set by `setkey()`.

If the argument *edflag* is zero, the argument is encoded, otherwise it is decoded.

ERRORS

Under the following conditions, these functions fail, and set `errno` to:

`ENOSYS` The functionality is not supported on this implementation.

USAGE

The return value of the function `crypt()` points to static data that are overwritten by each call.

LEVEL

Level 1.

Optional: the functionality of `crypt()`, `setkey()` and `encrypt()` may not be present in all implementations of the Base System. On implementations which do not support this functionality, calls to these functions will return with `errno` set to `ENOSYS`.

NAME

ctermid – generate filename for terminal

SYNOPSIS

```
#include <unistd.h>
#include <stdio.h>

char *ctermid(char *s);
```

DESCRIPTION

The function `ctermid()` generates the pathname of the controlling terminal for the current process and stores it in a string. Access to the file is not guarranteed.

If the argument *s* is a `null` pointer, the string is stored in an internal static area which will be overwritten at the next call to `ctermid()`. The address of the static area is returned. Otherwise, *s* is assumed to point to a character array of at least `L_ctermid` elements; the pathname is placed in this array and the value of *s* is returned.

RETURN VALUE

The function `ctermid()` returns an empty string if the pathname that would refer to the controlling terminal cannot be determined.

USAGE

The difference between the `ttyname()` routine and the function `ctermid()` is that the `ttyname()` routine must be passed a file descriptor and returns the name of the terminal associated with that file descriptor, whereas the function `ctermid()` returns the name of the controlling terminal for the current process.

SEE ALSO

ttyname(BA_LIB).

LEVEL

Level 1.

NAME

ctime, **localtime**, **gmtime**, **asctime**, **tzset** – convert date and time to string

SYNOPSIS

```
#include <time.h>
char *ctime(const time_t *clock);
struct tm *localtime(const time_t *clock);
struct tm *gmtime(const time_t *clock);
char *asctime(const struct tm *tm);
extern int daylight;
extern char *tzname[2];
void tzset(void);
```

DESCRIPTION

ctime, **localtime**, and **gmtime** accept arguments of type **time_t**, pointed to by *clock*, representing the time in seconds since 00:00:00 UTC, January 1, 1970. **ctime** returns a pointer to a 26-character string as shown below. Time zone and daylight savings corrections are made before the string is generated. The fields are constant in width:

```
Fri Aug 13 00:00:00 1993\n\0
```

localtime and **gmtime** return pointers to **tm** structures, described below. **localtime** corrects for the main time zone and possible alternate (''daylight savings'') time zone; **gmtime** converts directly to Coordinated Universal Time (UTC), which is the time the UNIX system uses internally.

asctime converts a **tm** structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the **tm** structure, are in the **time.h** header file.

The value of **tm_isdst** is positive if daylight savings time is in effect, zero if daylight savings time is not in effect, and negative if the information is not available. (Previously, the value of **tm_isdst** was defined as non-zero if daylight savings time was in effect.)

The external variable **timezone** contains the difference, in seconds, between UTC and local standard time. The external variable **daylight** indicates whether time should reflect daylight savings time. **timezone** defaults to 0 (UTC). The external variable **daylight** is non-zero if an alternate time zone exists. The time zone names are contained in the external variable **tzname**, which by default is set to:

```
char *tzname[2] = { "GMT", "   " };
```

These functions know about the peculiarities of this conversion for various time periods for the U.S.A. (specifically, the years 1974, 1975, and 1987). They will handle the new daylight savings time starting with the first Sunday in April, 1987.

tzset uses the contents of the environment variable **TZ** to override the value of the different external variables. It also sets the external variable **daylight** to zero if Daylight Savings Time conversions should never be applied for the time zone in use; otherwise, non-zero. **tzset** is called by **asctime** and may also be called by the user. See **environ**() for a description of the **TZ** environment variable.

SEE ALSO

getenv(BA_LIB), **mktime**(BA_LIB), **printf**(BA_LIB), **putenv**(BA_LIB), **setlocale**(BA_OS), **strftime**(BA_LIB), **time**(BA_OS),

LEVEL

Level 1.

NOTICES

The functions **ctime**, **localtime**, **fgmtime**, **tzset** and **asctime** are BA_LIB functions, and identical to the **ctime** BA_LIB page. **ctime_r**, **localtime_r** and **gmtime_r** are MT_LIB functions.

The return values for **ctime**, **localtime**, and **gmtime** point to static data whose content is overwritten by each call.

Setting the time during the interval of change from **timezone** to **altzone** or vice versa can produce unpredictable results. The system administrator must change the Julian start and end days annually.

Use the reentrant functions for multithreaded applications.

NAME

ctype: isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

SYNOPSIS

```
#include <ctype.h>

int isalpha(int c);

int isupper(int c);

int islower(int c);

int isdigit(int c);

int isxdigit(int c);

int isalnum(int c);

int isspace(int c);

int ispunct(int c);

int isprint(int c);

int isgraph(int c);

int iscntrl(int c);

int isascii(int c);
```

DESCRIPTION

These macros classify character-coded integer values. Each is a predicate returning non-zero for true, zero for false. The behavior of these macros, except `isascii()`, `isdigit()`, and `isxdigit()` is affected by the current locale [see setlocale(BA_OS)]. In the `"C"` locale, or in a locale where character type information is not defined, characters are classified according to the rules of the US-ASCII 7-bit coded character set.

The macro `isascii()` is defined on all integer values; the rest are defined only where the argument is an `int`, the value of which is representable as an `unsigned char`, or `EOF`, which is defined by the `<stdio.h>` header file and represents end-of-file.

`isalpha()` tests for any character for which `isupper()` or `islower()` is true, or any character that is one of an implementation-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, or `isspace()` is true. In the `"C"` locale, `isalpha()` returns true only for the characters for which `isupper()` or `islower()` is true.

`isupper()` tests for any character that is an upper-case letter or is one of an implementation-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, `isspace()`, or `islower()` is true. In the `"C"` locale, `isupper()` returns true only for the characters defined as upper-case ASCII characters.

`islower()` tests for any character that is a lower-case letter or is one of an implementation-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, `isspace()`, or `isupper()` is true. In the `"C"` locale, `islower()` returns true only for the characters defined as lower-case ASCII characters.

`isdigit()` tests for any decimal-digit character.

`isxdigit()` tests for any hexadecimal-digit character (`[0–9]`, `[A–F]` or `[a–f]`).

`isalnum()` tests for any character for which `isalpha()` or `isdigit()` is true (letter or digit).

`isspace()` tests for any space, tab, carriage-return, newline, vertical-tab or form-feed (standard white-space characters) or for one of an implementation-defined set of characters for which `isalnum()` is false. In the `"C"` locale, `isspace()` returns true only for the standard white-space characters.

`ispunct()` tests for any printing character which is neither a space nor a character for which `isalnum()` is true.

`isprint()` tests for any printing character, including space (`" "`).

`isgraph()` tests for any printing character, except space.

`iscntrl()` tests for any "control character" as defined by the character set.

`isascii()` tests for any ASCII character, code between `0` and `0177` inclusive.

Functions must exist for all the above defined macros. To get the function form, the macro name must be undefined (*e.g.* `#undef isdigit`).

RETURN VALUE

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

setlocale(BA_OS).

LEVEL

Level 1.

NAME

difftime – computes the difference between two calendar times

SYNOPSIS

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

DESCRIPTION

The function `difftime()` computes the difference between two calendar times. `difftime()` returns the difference (*time1* minus *time0*) expressed in seconds as a `double`.

USAGE

This function is provided because there are no general arithmetic properties defined for type `time_t`.

SEE ALSO

ctime(BA_LIB).

LEVEL

Level 1.

NAME

div, ldiv – compute the quotient and remainder

SYNOPSIS

```
#include <stdlib.h>

div_t div(int numer, int denom);

ldiv_t ldiv(long int numer, long int denom);
```

DESCRIPTION

The function `div()` computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. This function provides well-defined semantics for the signed integral division and remainder operations.

`div()` returns a structure of type `div_t` which includes the following members:

```
int  quot;     /* quotient */
int  rem; /* remainder */
```

`ldiv()` is similar to `div()`, except that the arguments and the members of the returned structure (which has type `ldiv_t`) all have type `long int`.

RETURN VALUE

If the result cannot be represented, the behavior is undefined; otherwise, *quotient* * *denom* + *remainder* will equal *numer*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient.

LEVEL

Level 1.

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

```
#include <stdlib.h>

double drand48(void);

double erand48(unsigned short xsubi[3]);

long lrand48(void);

long nrand48(unsigned short xsubi[3]);

long mrand48(void);

long jrand48(unsigned short xsubi[3]);

void srand48(long seedval);

unsigned short *seed48(unsigned short seed16v[3]);

void lcong48(unsigned short param[7]);
```

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

The functions `drand48()` and `erand48()` return non-negative double-precision floating-point values uniformly distributed over the interval $[0.0, 1.0)$.

The functions `lrand48()` and `nrand48()` return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

The functions `mrand48()` and `jrand48()` return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

The functions `srand48()`, `seed48()` and `lcong48()` are initialization entry points, one of which should be invoked before either `drand48()`, `lrand48()` or `mrand48()` are called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if `drand48()`, `lrand48()` or `mrand48()` are called without a prior call to an initialization entry point.) Functions `erand48()`, `nrand48()` and `jrand48()` do not require an initialization entry point to be called first. All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula:

$$X_{n+1} = (aX_n + c) \bmod m \qquad n \geq 0$$

The parameter `m` = 2^{48}; hence 48-bit integer arithmetic is performed. Unless `lcong48()`

and transformed into the returned value.

The functions `drand48()`, `lrand48()` and `mrand48()` store the last 48-bit X_i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions `erand48()`, `nrand48()` and `jrand48()` require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions `erand48()`, `nrand48()` and `jrand48()` allow separate modules of a large program to generate several independent streams of pseudo-random numbers. In other words, the sequence of numbers in each stream will not depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function `srand48()` sets the high-order 32-bits of X_i to the bits contained in its argument *seedval*. The low-order 16-bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function `seed48()` sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by `seed48()`, and a pointer to this buffer is the value returned by `seed48()`.

The initialization function `lcong48()` allows the user to specify the initial X_i, the multiplier value *a* and the addend value *c*. Argument array elements *param*`[0-2]` specify X_i, *param*`[3-5]` specify the multiplier *a*, and *param*`[6]` specifies the 16-bit addend *c*. After `lcong48()` has been called, a subsequent call to either `srand48()` or `seed48()` will restore the standard multiplier and addend values, *a* and *c*, specified above.

USAGE

The pointer returned by `seed48()`, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time. Use the pointer to get at and store the last X_i value and then use this value to reinitialize via `seed48()` when the program is restarted.

SEE ALSO

rand(BA_LIB).

LEVEL

Level 1

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf(double x);

double erfc(double x);
```

DESCRIPTION

The function `erf()` returns the error function of *x*, defined as follows:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

The function `erfc()` returns `1.0-erf(`*x*`)`.

RETURN VALUE

For both `erf()` and `erfc()`, if an input parameter is `NaN`, then the function will return `NaN` and set errno to `EDOM`.

USAGE

The function `erfc()` is provided because of the extreme loss of relative accuracy if `erf(`*x*`)` is called for large *x* and the result subtracted from `1.0`.

SEE ALSO

exp(BA_LIB).

LEVEL

Level 1.

NAME

exp, log, log10, pow, sqrt, cbrt – exponential, logarithm, power, root functions

SYNOPSIS

```
#include <math.h>

double exp(double x);

double log(double x);

double log10(double x);

double pow(double x, double y);

double sqrt(double x);

double cbrt(double x);
```

DESCRIPTION

The function `exp()` returns e^x.

The function `log()` returns the natural logarithm of *x*. The value of *x* must be positive.

The function `log10()` returns the base ten logarithm of *x*. The value of *x* must be positive.

The function `pow()` returns x^y. If *x* is zero, *y* must be non-negative. If *x* is negative, *y* must be an integer.

The function `sqrt()` returns the non-negative square root of *x*. The value of *x* may not be negative.

The function `cbrt()` returns the cube root of *x*.

RETURN VALUE

A macro `HUGE_VAL` will be defined by the `<math.h>` header file. This macro evaluates to a positive double expression, not necessarily representable as a float. On implementations that support the IEEE 754 standard, HUGE_VAL evaluates to $+\infty$.

If an input parameter is `NaN`, then all functions will return `NaN` and set errno to `EDOM`. The only exception is for `pow()`, which always returns 1 when its second argument is 0, regardless of the value of its first argument.

The function `exp()` returns `HUGE_VAL` when the correct value would overflow and sets `errno` to `ERANGE`. The function `exp()` returns `0` when the correct value would underflow and sets `errno` to `ERANGE`.

The functions `log()` and `log10()` will return an implementation-defined value (IEEE `NaN` or equivalent if available) and will set `errno` to `EDOM` when *x* is negative, and will return –`HUGE_VAL` and set `errno` to `ERANGE` when *x* is zero.

The function `pow()` will return an implementation-defined value (IEEE `NaN` or equivalent if available) and set `errno` to `EDOM` when the first argument is negative and the second is non-integral. When the first argument is 0 and the second argument is negative, finite, and an odd integer, `pow()` returns ±`HUGE_VAL`, according to the sign of the first argument and sets `errno` to `EDOM`. When the first argument is 0 and the second argument is negative, finite, and not an odd integer, `pow`

returns HUGE_VAL and sets errno to EDOM. The return value will be 1 with no error when both arguments are zero. The return value will be ±HUGE_VAL and errno will be set to ERANGE when the correct value would overflow. The return value will be 0 and errno will be set to ERANGE when the correct value would underflow.

On a system that supports the IEEE 754 standard, pow returns NAN and sets errno to EDOM when x is ±1 and y is ±∞.

The function sqrt() will return an implementation-defined value (IEEE NaN or equivalent if available) and set errno to EDOM when x is negative.

SEE ALSO

hypot(BA_LIB), hyperbolic(BA_LIB).

LEVEL

Level 1.

NAME

fattach – attach a STREAMS-based file descriptor to an object in the file system name space

SYNOPSIS

```
int fattach(int fildes, const char *path);
```

DESCRIPTION

The `fattach()` routine attaches a STREAMS-based file descriptor to an object in the file system name space, effectively associating a name with *fildes*. *fildes* must be a valid open file descriptor representing a STREAMS file. *path* is a pathname of an existing object and the process must have appropriate privileges or be the owner of the file and have write permissions. When the Enhanced Security Extension is implemented, *fildes* and *path* must have the same MAC level. All subsequent operations on *path* will operate on the STREAMS file until such time that the STREAMS file is detached from the node. A *fildes* can be attached to more than one *path*, that is, a stream can have several names associated with it.

The attributes of the named stream [see `stat(BA_OS)`] are initialized as follows: the permissions, user ID, group ID, and times are set to those of *path*, the number of links is set to 1, and the size and dev' set to those of the streams device associated with *fildes*. If any attributes of the named stream are subsequently changed (for example, chmod), the attributes of the underlying object are not affected.

RETURN VALUE

Upon successful completion, the `fattach()` routine returns a value of `0`; otherwise, a value of `–1` is returned and `errno` is set to indicate an error.

ERRORS

Under the following conditions, `fattach()` fails and sets `errno` to:

`EACCES`	if the user is the owner of *path* but does not have write permissions on *path* or if *fildes* is locked.
`EACCES`	if *fildes* and *path* do not have the same MAC level.
`EBADF`	if *fildes* is not a valid open file descriptor.
`ENOENT`	if *path* does not exist.
`ENOTDIR`	if a component of a path prefix is not a directory.
`EINVAL`	if *fildes* is not a STREAMS file.
`EPERM`	if the effective user ID is not the owner of *path* or a user with the appropriate privileges.
`EBUSY`	if *path* is currently a mount point or has a STREAMS file descriptor attached it.
`ENAMETOOLONG`	if the size of *path* exceeds `{PATH_MAX}`, or the component of a pathname is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
`ELOOP`	if too many symbolic links were encountered in translating *path*.

SEE ALSO

fdetach(BA_LIB), isastream(BA_LIB), streams (BA_DEV).

FUTURE DIRECTIONS

The `fattach()` routine may be enhanced in the future to enable a file descriptor that is not associated with a STREAMS-based file to be attached to an object in the file system name space.

LEVEL

Level 1.

NAME

fdetach – detach a name from a STREAMS-based file descriptor

SYNOPSIS

```
int fdetach(const char *path);
```

DESCRIPTION

The `fdetach()` routine detaches a STREAMS-based file descriptor from a name in the file system. *path* is the pathname of the object in the file system name space, which was previously attached [see fattach(BA_LIB)]. The user must be the owner of the file or be a user with the appropriate privileges. All subsequent operations on *path* will operate on the file system node and not on the STREAMS file. The permissions and status of the node are restored to the state the node was in before the STREAMS file was attached to it.

RETURN VALUE

Upon successful completion, the function `fdetach()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `fdetach()` fails and sets `errno` to:

`EPERM`	if the effective user ID is not the owner of *path* or is not a user with appropriate permissions.
`ENOTDIR`	if a component of the path prefix is not a directory.
`ENOENT`	if *path* does not exist.
`EINVAL`	if *path* is not attached to a STREAMS file.
`ENAMETOOLONG`	if the size of a pathname exceeds `{PATH_MAX}`, or pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
`ELOOP`	if too many symbolic links were encountered in translating *path*.

SEE ALSO

fattach(BA_LIB), streams(BA_DEV).

FUTURE DIRECTIONS

`fdetach()` may be enhanced in the future to enable a file descriptor that is not associated with a STREAMS-based file to be detached from a node.

LEVEL

Level 1.

NAME

floor, ceil, fmod, remainder, fabs – floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>
double floor(double x);
double ceil(double x);
‡ double fmod(double x, double y);
double remainder(double x, double y);
double fabs(double x);
```

DESCRIPTION

The function `floor()` returns the largest integral value not greater than *x*.

The function `ceil()` returns the smallest integral value not less than *x*.

The function `fmod()` returns the floating point remainder $f = x - my$ when y is non-zero, where m is the integral value chosen so that f has the same sign as x and $|f| < |y|$.

The function `remainder()` returns the floating point remainder $r = x - ny$ when y is non-zero. The value n is the integral value nearest the exact value x/y; when $|n - x/y| = ½$, the value n is chosen to be even.

The function `fabs()` returns $|x|$, the absolute value of x.

RETURN VALUE

If an input parameter is `NaN`, then the function will return `NaN` and set errno to `EDOM`.

When y is zero the functions `fmod()` and `remainder()` will return an implementation-defined value (IEEE `NaN` or equivalent if available) and set `errno` to `EDOM`.

On a system that supports the IEEE 754 standard, if the value of x for `fmod()` or `remainder()` is +-∞, these functions will return IEEE `NaN` and set errno to `EDOM`.

SEE ALSO

abs(BA_LIB).

LEVEL

Level 1. fmod() function Level 2, effective 9/30/89.

NAME

fmtmsg – display a message in the standard format on standard error and the system console

SYNOPSIS

```
#include <fmtmsg.h>

int fmtmsg(long classification, const char *label, int severity,
     const char *text, const char *action, const char *tag);
```

DESCRIPTION

The function `fmtmsg()` can be used to display messages in standard format instead of the traditional `printf()` interface. `fmtmsg()` in conjunction with `gettext()` provides a simple interface for producing language-independent applications.

Based on a message's classification component, the function `fmtmsg()` either writes a formatted message to standard error, the console, or to both.

A formatted message consists of up to five standard components as defined below. The component, *classification*, is not part of the standard message displayed to the user, but defines the source of the message and directs the display of the formatted message.

classification

Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and the system console).

major classifications

Identifies the source of the condition. Identifiers are: `MM_HARD` (hardware), `MM_SOFT` (software), and `MM_FIRM` (firmware).

message source subclassifications

Identifies the type of software in which the problem is detected. Identifiers are: `MM_APPL` (application), `MM_UTIL` (utility), and `MM_OPSYS` (operating system).

display subclassifications

Indicates where the message is to be displayed. Identifiers are: `MM_PRINT` to display the message on the standard error stream, `MM_CONSOLE` to display the message on the system console. One or both identifiers may be used.

status subclassifications

Indicates whether the application will recover from the condition. Identifiers are: `MM_RECOVER` (recoverable) and `MM_NRECOV` (non-recoverable).

An additional identifier, MM_NULLMC, indicates that no classification component is supplied for the message.

label Identifies the source of the message. The format is two fields separated by a colon. The first field is up to 10 characters, the second is up to 14 characters. Suggested usage is that *label* identifies the package in which the application resides as well as the program or application name. For example, the *label* UX:cat indicates the operating system package and the cat application.

severity
Indicates the seriousness of the condition. Identifiers for the standard levels of *severity* are:

MM_HALT
indicates that the application has encountered a severe fault and is halting. Produces the print string HALT.

MM_ERROR
indicates that the application has detected a fault. Produces the print string ERROR.

MM_WARNING
indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the print string WARNING.

MM_INFO
provides information about a condition that is not in error. Produces the print string INFO.

MM_NOSEV
indicates that no severity level is supplied for the message. Describes the error condition that produced the message. The text string is not limited to a specific size.

text Describes the error condition that produced the message. If the text string is null then a message will be issued stating that no text has been provided.

action Describes the first step to be taken in the error-recovery process. fmtmsg() precedes the action string with the prefix: TO FIX:. The *action* string is not limited to a specific size.

tag An identifier which references on-line documentation for the message. Suggested usage is that *tag* includes the *label* and a unique identifying number. A sample *tag* is UX:cat:146.

Environment Variables

There are two environment variables that control the behavior of fmtmsg(): MSGVERB (message verbosity) and SEV_LEVEL (severity level). SEV_LEVEL can be used in shell scripts or set in the user's shell. MSGVERB can be set by the administrator in the /etc/profile for the system. Users can override the system-set MSGVERB by resetting MSGVERB in their own .profile files or by changing the value in their current shell session.

MSGVERB tells fmtmsg() which message components it is to select when writing messages to standard error. The value of MSGVERB is a colon-list of optional keywords. MSGVERB can be set as follows:

```
MSGVERB=[keyword[:keyword[:...]]]
export MSGVERB
```

Valid *keywords* are: label, severity, text, action, and tag. If MSGVERB contains a keyword for a component and the component's value is not the component's null value, fmtmsg() includes that component in the message when writing the message to standard error. If MSGVERB does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If MSGVERB is not defined, if its value is the null-string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, fmtmsg() selects all components.

MSGVERB affects only which components are selected for display to standard error. All message components are included in console messages.

SEV_LEVEL defines severity levels and associates print strings with them for use by fmtmsg(). The standard severity levels shown below cannot be modified. Additional severity levels can be defined, redefined, and removed.

```
0   (no severity is used)
1   HALT
2   ERROR
3   WARNING
4   INFO
```

SEV_LEVEL can be set as follows:

```
SEV_LEVEL=[description[:description[:...]]]
export SEV_LEVEL
```

The format of *description* is a three-field comma list as follows:

description=severity_keyword,level,printstring

where

severity_keyword
: is not used by the fmtmsg() function; it is used by the fmtmsg command [see fmtmsg(BU_CMD)].

level
: is a character string that evaluates to a positive integer (other than 0, 1, 2, 3, or 4, which are reserved for the standard severity levels). The command fmtmsg uses *severity-keyword* and passes *level* onto fmtmsg().

printstring
: is the character string used by fmtmsg() in the standard message format whenever the severity value *level* is used.

If SEV_LEVEL is not defined, or if its value is null, no severity levels other than the defaults are available. If a *description* in the colon list is not a three-field comma list, or, if the second field of a comma list does not evaluate to a positive integer, that *description* in the colon list is ignored.

Use in Applications

One or more message components may be systematically omitted from messages generated by an application by using the null value of the argument for that component. The table below indicates the null values and identifiers for `fmtmsg()` arguments.

Argument	Type	Null-Value	Identifier
label	`char*`	`(char*)NULL`	`MM_NULLLBL`
severity	`int`	`0`	`MM_NULLSEV`
class	`long`	`0L`	`MM_NULLMC`
text	`char*`	`(char*)NULL`	`MM_NULLTXT`
action	`char*`	`(char*)NULL`	`MM_NULLACT`
tag	`char*`	`(char*)NULL`	`MM_NULLTAG`

Another means of systematically omitting a component is by omitting the component keyword(s) when defining the `MSGVERB` environment variable (see **Environment Variables**).

ERRORS

The exit codes for `fmtmsg()` are the following:

`MM_OK` = the function succeeded
`MM_NOTOK` = the function failed completely
`MM_NOMSG` = the function was unable to generate a message on standard error, but otherwise succeeded.
`MM_NOCON` = the function was unable to generate a console message, but otherwise succeeded.

EXAMPLE

Example 1:

The following example of `fmtmsg()`:

```
fmtmsg(MM_PRINT, "UX:cat", MM_ERROR, "illegal option",
"refer to cat in user's reference manual", "UX:cat:001")
```

produces a complete message in the standard message format:

```
UX:cat: ERROR: illegal option
TO FIX: refer to cat in user's reference manual UX:cat:001
```

Example 2:

When the environment variable `MSGVERB` is set as follows:

```
MSGVERB=severity:text:action
```

and the Example 1 is used, `fmtmsg()` produces:

```
ERROR: illegal option
TO FIX: refer to cat in user's reference manual
```

Example 3:

When the environment variable SEV_LEVEL is set as follows:

```
SEV_LEVEL=note,5,NOTE
```

the following call to `fmtmsg()`:

```
fmtmsg(MM_PRINT | MM_UTIL, "UX:cat", 5, "cannot open file",
"specify correct file name", "UX:cat:002")
```

produces:

```
UX:cat: NOTE: cannot open file
TO FIX: specify correct file name UX:cat(1):002
```

SEE ALSO

fmtmsg(BU_CMD), gettxt(BA_LIB), printf(BA_LIB).

FUTURE DIRECTIONS

This interface is to be removed when the three-year waiting period has expired. It is replaced by `pfmt`.

LEVEL

Level 2: April 1991.

NAME

fnmatch – match filename or pattern

SYNOPSIS

```
#include <fnmatch.h>

int fnmatch(const char *pattern, const char *string, int flags);
```

DESCRIPTION

fnmatch is part of the X/Open Portability Guide Issue 4 optional POSIX2 C-Language Binding feature group.

Return Values

fnmatch returns **FNM_NOSYS** and sets **errno** to **ENOSYS**.

USAGE

Administrator.

SEE ALSO

glob(BA_LIB), **wordexp**(BA_LIB)

LEVEL

Level 1.

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

```
#include <math.h>
double frexp(double value, int *eptr);

‡ double ldexp(double value, int exp);

double modf(double value, double *iptr);
```

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the significand x is in the range $0.5 \leq | x | < 1.0$ and the exponent n is an integer. The function `frexp()` returns the significand of *value* and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by `frexp()` are zero.

The function `ldexp()` returns the quantity $value * 2^{exp}$.

The function `modf()` returns the fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*. Both the fractional and integral parts have the same sign as *value*.

RETURN VALUE

A macro `HUGE_VAL` will be defined by the `<math.h>` header file. This macro evaluates to a positive double expression, not necessarily representable as a float. On implementations that support the IEEE 754 standard, `HUGE_VAL` evalutates to $+\infty$.

If the correct value would overflow, `ldexp()` will return $\pm$`HUGE_VAL` (according to the sign of *value*) and set `errno` to `ERANGE`.

If the correct value would underflow, the function `ldexp()` returns `0` and sets `errno` to `ERANGE`.

If an input parameter is `NaN`, then the function will return `NaN` and set errno to `EDOM`.

SEE ALSO

exp(BA_LIB), scalb(BA_LIB).

LEVEL

Level 1. ldexp() is Level 2, effective 9 / 30 / 89.

NAME

ftok – standard interprocess communication package

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

DESCRIPTION

ftok returns a key based on *path* and *id* that is usable in subsequent **msgget**(KE_OS), **semget**(KE_OS, and **shmget**(KE_OS) system calls. *path* must be the path name of an existing file that is accessible to the process. *id* is a character that uniquely identifies a project. Note that **ftok** will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

If the file whose *path* is passed to **ftok** is removed when keys still refer to the file, future calls to **ftok** with the same *path* and *id* will return an error. If the same file is recreated, then **ftok** is likely to return a different key than it did the original time it was called.

SEE ALSO

msgget(KE_OS), **semget**(KE_OS), **shmget**(KE_OS).

RETURN VALUE

ftok returns **(key_t) -1** if *path* does not exist or if it is not accessible to the process.

LEVEL

Level 2, April 1991.

NAME

ftw, nftw – walk a file tree

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <ftw.h>
int ftw(const char *path,
   int (*fn) (const char *, const struct stat *, int), int depth);
int nftw(const char *path,
   int (*fn)(const char *, const struct stat *, int, struct FTW*),
   int depth, int flags);
```

DESCRIPTION

The function `ftw()` descends the directory hierarchy rooted in *path*. For each node in the hierarchy, the function `ftw()` calls a user-defined function *fn()* passing it three arguments. The first argument passed is a character pointer to a null-terminated string containing the name of the node. The second argument passed to *fn()* is a pointer to a `stat` structure [see stat(BA_OS)] containing information about the node, and the third argument passed is an integer. Possible values of the parameter, defined by the `<ftw.h>` header file, are `FTW_F` for a file, `FTW_D` for a directory, `FTW_DNR` for a directory that cannot be read and `FTW_NS` for an object for which `stat()` could not successfully be executed. If the integer is `FTW_DNR`, descendants of that directory will not be processed. If the integer is `FTW_NS`, the contents of the `stat` structure are undefined.

The function `nftw()` works similarly as `ftw()` except that it takes on an additional argument *flags*. The flags field is used to specify:

`FTW_PHYS` Physical walk, does not follow symbolic links. Otherwise, `nftw()` will follow links but will not walk down any path that crosses itself.

`FTW_MOUNT` The walk will not cross a mount point.

`FTW_DEPTH` All subdirectories will be visited before the directory itself.

`FTW_CHDIR` The walk will change to each directory before reading it.

The function `nftw()` calls *fn()* with four arguments at each file and directory. The first argument is the pathname of the object, the second is a pointer to the `stat` buffer, and the third is an integer giving additional information as follows:

`FTW_F` The object is a file.

`FTW_D` The object is a directory.

`FTW_DP` The object is a directory and subdirectories have been visited.

`FTW_SL` The object is a symbolic link.

`FTW_DNR` The object is a directory that cannot be read. *fn()* will not be called for any of its descendants.

`FTW_NS` `stat()` failed on the object because of lack of appropriate permission. The `stat` buffer passed to *fn()* is undefined. `stat()` failure for any reason is considered an error and `nftw()` will return `-1`.

The fourth argument is a `struct FTW` which contains the following members:

```
int base;
int level;
```

The value of `base` is the offset into the pathname of the object; this pathname is passed as the first argument to *fn()*. The value of `level` indicates depth relative to the root of of the walk, where the root level has a value of zero.

The function `ftw()` visits a directory before visiting any of its descendants.

Both functions use one file descriptor for each level in the tree. The argument *depth* limits the number of file descriptors so used. The argument *depth* should be in the range of `1` to `{OPEN_MAX}`. The function `ftw()` will run more quickly if *depth* is at least as large as the number of levels in the tree. When the function `ftw()` returns it closes any file descriptors it has opened but not those opened by the user supplied function *fn()*.

RETURN VALUE

The tree traversal continues until the tree is exhausted, an invocation of *fn()* returns a non-zero value or some error is detected within `ftw()` (such as an I/O error). If the tree is exhausted, the function `ftw()` returns `0`. If the function *fn()* returns a non-zero value, the function `ftw()` stops its tree traversal and returns whatever value was returned by the function *fn()*.

If the function `ftw()` encounters an error other than `EACCES` (see `FTW_DNR` and `FTW_NS` above), it returns `–1` and `errno` is set to the type of error. The external variable `errno` may contain the error values that are possible when a directory is opened [see open(BA_OS)] or when the `stat()` routine is executed on a directory or file.

ERRORS

Under the following conditions, the function `ftw()` fails and sets `errno` to:

`EACCES` if a component of the *path* prefix denies search permission or read permission is denied for *path*, and `fu()` returns `-1` and does not reset errno.

`ENAMETOOLONG` if the length of the *path* string exceeds `{PATH_MAX}`, or a pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

`ENOENT` if the *path* argument points to the name of a file which does not exist or points to an empty string.

`ENOTDIR` if a component of *path* is not a directory.

SEE ALSO

stat(BA_OS), malloc(BA_OS).

LEVEL

Level 1.

NAME

fwprintf, **wprintf**, **swprintf** – print formatted wide/multibyte character output

SYNOPSIS

```
#include <wchar.h>

int fwprintf(FILE *strm, const wchar_t *format, .../* args */);

int swprintf(wchar_t *s, size_t maxsize, const wchar_t *format,
      .../* args */);

int wprintf(const wchar_t *format, .../* args */);
```

DESCRIPTION

Each of these functions converts, formats, and outputs its *args* under control of the wide character string *format*. Each function returns the number of wide/multibyte characters transmitted (not including the terminating null wide character in the case of **swprintf**) or a negative value if an output error was encountered.

fwprintf places multibyte output on *strm*.

wprintf places multibyte output on the standard output stream **stdout**.

swprintf places wide character output, followed by a null wide character (**\0**), in consecutive wide characters starting at *s*, limited to no more than *maxsize* wide characters. If more than *maxsize* wide characters were requested, the output array will contain exactly *maxsize* wide characters, with a null wide character being the last (when *maxsize* is nonzero); a negative value is returned.

The *format* consists of zero or more ordinary wide characters (not **%**) which are directly copied to the output, and zero or more conversion specifications, each of which is introduced by the a **%** and results in the fetching of zero or more associated *args*.

Each conversion specification takes the following general form and sequence:

%[*pos***$][** *flags* **][** *width* **][.**_prec_ **][** *size* **]***fmt*

*pos***$** An optional entry, consisting of one or more decimal digits followed by a **$** character, that specifies the number of the next *arg* to access. The first *arg* (just after *format*) is numbered 1. If this entry is not present, the *arg* following the most recently used *arg* will be accessed.

flags Zero or more wide characters that change the meaning of the conversion specification. The *flag* characters and their meanings are:

- **-** The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)
- \+ The result of a signed conversion will always begin with a sign (+ or **-**). (It will begin with a sign only when a negative value is converted if this flag is not specified.)
- *space* If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide characters, a space will be prefixed to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

The value is to be converted to an alternate form, depending on the *fmt* wide character:

a, **A**, **e**, **E**, **f**, **F**, **g**, **G**
The result will contain a decimal point wide character, even if no digits follow. (Normally, the decimal point wide character is only present when fractional digits are produced.)

b, **B** A nonzero result will have **0b** or **0B** prefixed to it.

g, **G** Trailing zero digits will not be removed from the result, as they normally are.

o The precision is increased (only when necessary) to force a zero as the first digit.

x, **X** A nonzero result will have **0x** or **0X** prefixed to it.

For other conversions, the behavior is undefined.

0 For all numeric conversions (**a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**, **b**, **B**, **d**, **i**, **o**, **u**, **x** and **X**), leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **–** flags both appear, the **0** flag will be ignored. For the integer numeric conversions (**b**, **B**, **d**, **i**, **o**, **u**, **x** and **X**), if a precision is specified, the **0** flag will be ignored. For other conversions, the behavior is undefined.

´ (an apostrophe) The nonfractional portion of the result of a decimal numeric conversion (**d**, **i**, **u**, **f**, **F**, **g** and **G**) will be grouped by the current locale's thousands' separator wide character.

width An optional entry that consists of either one or more decimal digits, or an asterisk (*****), or an asterisk followed by one or more decimal digits and a **$**. It specifies the minimum field width: If the converted value has fewer wide / multibyte characters than the field width, it will be padded (with space by default) on the left or right (see the above *flags* description) to the field width.

.*prec* An optional entry that consists of a period (**.**) that precedes either zero or more decimal digits, or an asterisk (*****), or an asterisk followed by one or more decimal digits and a **$**. It specifies a value that depends on the *fmt* wide character:

a, **A**, **e**, **E**, **f**, **F**
It specifies the number of fractional digits (those after the decimal point wide character). For the hexadecimal floating conversions (**a** and **A**), the number of fractional digits is just sufficient to produce an exact representation of the value (trailing zero digits are removed); for the other conversions, the default number of fractional digits is 6.

b, **B**, **d**, **i**, **o**, **u**, **x**, **X**
It specifies the minimum number of digits to appear. The default minimum number of digits is 1.

g, **G** It specifies the maximum number of significant digits. The default number of significant digits is 6.

s, **S** It specifies the maximum number of wide/multibyte characters to output. The default is to take all elements up to the null terminator (the entire string).

If only a period is specified, the precision is taken to be zero. For other conversions, the behavior is undefined.

size An optional **h**, **l** (ell), or **L** that specifies other than the default argument type, depending on the *fmt* character:

a, **A**, **e**, **E**, **f**, **F**, **g**, **G**
The default argument type is **double**; an **l** is ignored for compatibility with the **scanf** functions (a **float** *arg* will have been promoted to **double**); an **L** causes a **long double** *arg* to be converted.

b, **B**, **o**, **u**, **x**, **X**
The default argument type is **unsigned int**; an **h** causes the **unsigned int** *arg* to be narrowed to **unsigned short** before conversion; an **l** causes an **unsigned long** *arg* to be converted.

c The default argument type is **int** which is converted to a wide character as if by calling **btowc** before output; an **l** causes a **wchar_t** *arg* to be output. **lc** is a synonym for **C**.

d, **i** The default argument type is **int**; an **h** causes the **int** *arg* to be narrowed to **short** before conversion; an **l** causes a **long** *arg* to be converted.

n The default argument type is pointer to **int**; an **h** changes it to be a pointer to **short**, and **l** to pointer to **long**.

s The default argument type is pointer the first element of a character array; an **l** changes it to be a pointer to the first element of a **wchar_t** array. **ls** is a synonym for **S**.

If a *size* appears other than in these combinations, the behavior is undefined.

fmt A conversion wide character (described below) that shows the type of conversion to be applied.

When a field width or precision includes an asterisk (*****), an **int** *arg* supplies the width or precision value, and is said to be ''indirect''. A negative indirect field width value is taken as a **–** flag followed by a positive field width. A negative indirect precision value will be taken as zero. When an indirect field width or precision includes a **$**, the decimal digits similarly specify the number of the *arg* that supplies the field width or precision. Otherwise, an **int** *arg* following the most recently used *arg* will be accessed for the indirect field width, or precision, or both, in that order; the *arg* to be converted immediately follows these. Thus, if a conversion specification includes *pos***$** as well as a **$**-less indirect field width, or precision, or both, *pos* is taken to be the number of the **int** *arg* used for the first **$**-less indirection, not the *arg* to be converted.

When numbered argument specifications are used, specifying the *N*th argument requires that all the preceding arguments, from the first to the (*N*–1)th, be specified at least once, in a consistent way, in the format string.

The conversion wide characters and their meanings are:

a, **A** The floating *arg* is converted to hexadecimal floating notation in the style **[–]0x***h***.***hhh***p*±*d*. The binary exponent of the converted value (*d*) is one or more decimal digits. The number of fractional hexadecimal digits *h* is equal to the precision. If the precision is missing, the result will have just enough digits to represent the value exactly. The value is rounded when fewer fractional digits is specified. If the precision is zero and the **#** flag is not specified, no decimal point wide character appears. The single digit to the left of the decimal point character is nonzero for normal values. The **A** conversion specifier produces a value with **0X** and **P** instead of **0x** and **p**.

b, **B**, **o**, **u**, **x**, **X**
The unsigned integer *arg* is converted to unsigned binary (**b** and **B**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** and **X**). The **x** conversion uses the letters **abcdef** and the **X** conversion uses the letters **ABCDEF**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

c The integer *arg* is converted to a wide character as if by calling **btowc**, and the resulting wide character is output.

C, **lc** The wide character **wchar_t** *arg* is output.

d, **i** The integer *arg* is converted to signed decimal. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

e, **E** The floating *arg* is converted to the style **[–]***d***.***ddd***e**±*dd*, where there is one digit before the decimal point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal point wide character appears. The value is rounded to the appropriate number of digits. The **E** conversion wide character will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

f, **F** The floating *arg* is converted to decimal notation in the style **[–]***ddd***.***ddd*, where the number of fractional digits is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal point wide character appears. If a decimal point wide character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

g, **G** The floating *arg* is converted in style **e** or **f** (or in style **E** or **F** in the case of a **G** conversion wide character), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted; style **e** (or **E**) will be used only if the exponent resulting from the conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point wide character appears only if it is followed by a digit.

n The *arg* is taken to be a pointer to an integer into which is written the number of wide/multibyte characters output so far by this call. No argument is converted.

p The *arg* is taken to be a pointer to **void**. The value of the pointer is converted to an sequence of printable wide characters, which matches those read by the **%p** conversion of the **fwscanf**(BA_LIB) functions.

s The *arg* is taken to be a pointer to the first element of an array of characters. Multibyte characters from the array are output up to (but not including) a terminating null character; if a precision is specified, no more than that many wide/multibyte characters are output. If a precision is not specified or is greater than the size of the array, the array must contain a terminating null character. (A null pointer for *arg* will yield undefined results.)

S, **ls** The *arg* is taken to be a pointer to the first element of an array of **wchar_t**. Wide characters from the string are output until a null wide character is encountered or the number of wide/multibyte characters given by the precision wide would be surpassed. If the precision specification is missing, it is taken to be infinite. In no case will a partial wide/multibyte character be output.

% Output a **%**; no argument is converted.

If the form of the conversion specification does not match any of the above, the results of the conversion are undefined. Similarly, the results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are ignored.

If a floating-point value represents an infinity, the output is [±]*inf*, where *inf* is **infinity** or **INFINITY** when the field width or precision is at least 8 and **inf** or **INF** otherwise, the uppercase versions used only for a capitol conversion wide character. Output of the sign follows the rules described above.

If a floating-point value has the internal representation for a NaN (not-a-number), the output is [±]*nan*[**(***m***)**]. Depending on the conversion character, *nan* is similarly either **nan** or **NAN**. If the represented NaN matches the architecture's default, no **(***m***)** will be output. Otherwise *m* represents the bits from the significand in hexadecimal with **abcdef** or **ABCDEF** used, depending on the case of the conversion wide character. Output of the sign follows the rules described above.

Otherwise, the locale's decimal point wide character will be used to introduce the fractional digits of a floating-point value.

A nonexistent or small field width does not cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result. Multibyte characters generated on streams (**stdout** or *strm*) are printed as if the **putc** function had been called repeatedly.

Errors

These functions return the number of wide/multibyte characters transmitted (not counting the terminating null wide character for **swprintf** and **vswprintf**), or return a negative value if an error was encountered.

USAGE

To print a date and time in the form ''Sunday, July 3, 10:02,'' where **weekday** and **month** are pointers to null-terminated strings:

```
wprintf(L"%s, %s %i, %d:%.2d",
        weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
wprintf(L"pi = %.5f", 4 * atan(1.0));
```

The following two calls to **wprintf** both produce the same result of **10 10 00300 10**:

```
wprintf(L"%d %1$d %.*d %1$d", 10, 5, 300);
wprintf(L"%d %1$d %3$.*2$d %1$d", 10, 5, 300);
```

SEE ALSO

printf(BA_LIB), putc(BA_LIB), scanf(BA_LIB) setlocale(BA_LIB), stdio(BA_LIB), write(BA_OS)

LEVEL

Level 1.

NAME

fwscanf, **wscanf**, **swscanf** – convert formatted wide/multibyte character input

SYNOPSIS

```
#include <wchar.h>

int fwscanf(FILE *stream, const wchar_t *format, ...);

int wscanf(const wchar_t *format, ...);

int swscanf(const wchar_t *s, const wchar_t *format, ...);
```

DESCRIPTION

fwscanf reads input from the stream pointed to by **stream**, under control of the wide string pointed to by **format** that specifies admissible input sequences and how they are converted for input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while the arguments remain, the excess arguments are evaluated but are otherwise ignored.

wscanf reads input to the stream in the same manner as **fwscanf**, with the argument **stdin** interposed before the arguments to **wscanf**.

swscanf reads input to the stream in the same manner as **fwscanf**, except that the argument **s** specifies a wide string from which the generated input is read, rather than converting multibyte characters from a stream. Also, the detection of wide or multibyte encoding errors may differ. If the end of the wide string is reached, it behaves the same as when an end-of-file is encountered for **fwscanf**. If copying takes place between objects that overlap, the behavior is undefined.

The format is composed of zero or more directives which include:

One or more white space wide characters

Ordinary wide characters (not % or white space)

Conversion specifications (all wide characters which are members of the basic character set).

Each conversion specification is introduced by the wide character **%** and followed by:

An optional assignment-suppressing wide character *****.

An optional nonzero decimal integer that specifies the maximum field width.

An optional **h**, **l** or **L** indicating the size of the receiving object. The conversion specifiers **d**, **i**, and **n** are preceded by **h** if the corresponding argument is a pointer to **short int** instead of a pointer to **int**, or by **l** if it is a pointer to **long int**. The conversion specifiers **b**, **o**, **u** and **x** are preceded by **h** if the corresponding argument is a pointer to **unsigned short int** instead of a pointer to **unsigned int**, or by **l** if it is a pointer to an **unsigned long int**. The conversion specifiers **a**, **e**, **f** and **g** are preceded by **l** if the corresponding argument is a pointer to **double** rather than a pointer to **float** or by **L** if it is a pointer to **long double**. The conversion specifiers **c**, **s** and **[...]** are preceded by **l** if the corresponding argument is a pointer to **wchar_t** instead of a pointer to **character**. **lc** and **ls** are synonyms for **C** and **S** respectively. If an **h**, **l** or **L** appears with any other conversion

specifier, the behavior is undefined.

A wide character that specifies the type of conversion to be applied.

fwscanf executes each directive of the format in turn. If a directive fails, the function returns. Failures can be input failures if an encoding error occurs or if input characters are unavailable. Failures can also be matching failures if there is inappropriate input.

A directive comprised of white space wide characters is executed by reading input up to the first non white-space character which remains unread, or until no more wide characters can be read.

A directive that is an ordinary wide character is executed by reading the next wide character of the stream. If the wide character differs from the directive, the directive fails, and the differing and next wide characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed as follows:

1 Input white-space wide characters, as specified by the **iswspace** function, are skipped unless the specification includes a **c** or **n** specifier.

2 An input item is read from the stream unless the specification includes an **n** specifier. An input item is defined as the longest matching sequence of input wide characters unless that exceeds a specified field width. The first wide character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails. This condition is a matching failure, unless an error prevented input from the stream, which causes an input failure.

3 Except for a **%** specifier, the input item is converted to a type appropriate to the conversion specifier. This also applies to an **n** directive for the count of wide characters. If the input item is not a matching sequence, the execution of the directive fails. This constitutes a matching failure. Unless assignment suppression is indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following section lists the valid conversion specifiers and their meanings:

d Matches an optionally signed decimal integer whose format is the same as expected for the subject sequence of the **wcstol** function with the value **10** for the **base** argument. The corresponding argument is a pointer to an integer.

i Matches an optionally signed integer whose format is the same as expected for the subject sequence of the **wcstol** function with the value **0** for the **base** argument. The corresponding argument is a pointer to an integer.

b Matches an optionally signed binary integer whose format is the same as expected for the subject sequence of the **wcstoul** function with the value **2** for the **base** argument. The corresponding argument is a pointer to an integer.

o Matches an optionally signed octal integer whose format is the same as expected for the subject sequence of the **wcstoul** function with the value **8** for the **base** argument. The corresponding argument is a pointer to an integer.

u Matches an optionally signed decimal integer whose format is the same as expected for the subject sequence of the **wcstoul** function with the value **10** for the **base** argument. The corresponding argument is a pointer to an unsigned integer.

x Matches an optionally signed hexadecimal integer whose format is the same as expected for the subject sequence of the **wcstoul** function with the value **16** for the **base** argument. The corresponding argument is a pointer to an unsigned integer.

a,e,f,g
Matches an optionally floating point number whose format is the same as expected for the subject sequence of the **wcstod** function. The corresponding argument is a pointer to a floating point number.

s Matches a sequence of non-white-space wide/multibyte characters. The corresponding argument is a pointer to the initial element of an array of **wchar_t** type large enough to accept the sequence and a terminating null wide character that is added automatically.

c Matches a sequence of wide/multibyte characters of the number specified by the field width, or **1** if no field width is present in the directive. The corresponding argument is a pointer to the initial element of an array of **wchar_t** type large enough to accept the sequence. No null wide character is added.

C,lc Matches a sequence of wide/multibyte characters of the number specified by the field width (1 if no width is present in the directive). The corresponding argument should be a pointer to the initial element of a **wchar_t** array large enough to accept the sequence of wide characters. No null wide character is added. The normal skip over white space is suppressed.

S,ls Matches a sequence of wide/multibyte characters, optionally delimited by white-space wide/multibyte characters. The corresponding argument should be a pointer to the initial element of a **wchar_t** array large enough to accept the sequence of wide characters and a terminating null wide character, which will be added automatically.

p Matches an implementation-defined set of sequences that are the same as the set of sequences that are produced by the **%p** conversion of **fwprintf**. The corresponding argument is a pointer to **void**. The interpretation of the input is implementation defined. If the input item is a value converted earlier during the same program execution, the pointer that results compares equally to that value. Otherwise the behavior of **%p** is undefined.

n No input is consumed. The corresponding argument is a pointer to an integer into which is written the number of wide/multibyte characters read so far from the input stream written into it. Execution of a **%n** directive does not increment the assignment count returned at the completion of execution of this function.

[...] Matches a nonempty sequence of wide/multibyte characters from a set of expected wide characters (the *scanset*) as designated by the wide characters between the brackets (the *scanlist*), see below. The corresponding argument should be a pointer to the initial element of a character array large enough to accept the generated multibyte sequence and a terminating null character, which will be added automatically.

l[...]
Matches a nonempty sequence of wide/multibyte characters from a set of expected wide characters (the *scanset*) as designated by the wide characters between the brackets (the *scanlist*), see below. The corresponding argument should be a pointer to the initial element of a **wchar_t** array large enough to accept the sequence of wide characters and a terminating null wide character, which will be added automatically.

% Matches a single **%**. No conversion or assignment occurs. The complete conversion specification is **%%**.

For **[...]** and **l[...]**, the conversion specifier includes all subsequent characters in the the *format* string, up to and including the matching right bracket (**]**). The characters between the brackets (the *scanlist*) comprise the scanlist, unless the character after the left braket is a circumflex (^), in which case the scanlist contains all characters that do not appear in the scanlist and the right bracket. If the conversion specifier begins with **[]** or **[^]**, the right bracket character is in the scanlist and the next character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.

If a conversion specification is invalid, the behavior is undefined. The conversion specifiers **A, E, G** and **X** are also valid and behave the same as **a, e, g** and **x** respectively.

Errors

fwscanf, **wscanf** and **swscanf** return the number of wide characters transmitted or return a negative value if an error was encountered.

SEE ALSO

printf(BA_LIB), putc(BA_LIB), scanf(BA_LIB), setlocale(BA_LIB), stdio(BA_LIB), write(BA_OS)

LEVEL

Level 1.

NAME

get_t_errno, set_t_errno – get/set t_errno value

SYNOPSIS

```
#include <xti.h>

int get_t_errno(void)

int set_t_errno(int)
```

DESCRIPTION

The **get_t_errno** and **set_t_errno** functions are used in TLI/XTI multi-threaded applications to set and return the value in **t_errno**.

These functions are required by applications compiled with the **_REENTRANT** flag if the user needs to set the thread-specific version of **t_errno**.

USAGE

While **get_t_errno** and **set_t_errno** are designed for use in multi-threaded applications, they are available for used in non-reentrant code and may be incorporated if a need is anticipated to convert to reentrant code later on.

NAME

getc, **getchar**, **fgetc**, **getw** – get character or word from a stream

SYNOPSIS

```
#include <stdio.h>

int getc(FILE *stream);

int getchar(void);

int fgetc(FILE *stream);

int getw(FILE *stream);
```

DESCRIPTION

getc returns the next character (that is, byte) from the named input *stream* as an **unsigned char** converted to an **int**. It also moves the file pointer, if defined, ahead one character in *stream*. **getchar** is defined as **getc(stdin)**. **getc** and **getchar** are macros.

fgetc behaves like **getc**, but is a function rather than a macro. **fgetc** runs more slowly than **getc**, but it takes less space per invocation and its name can be passed as an argument to a function.

getw returns the next word (that is, integer) from the named input *stream*. **getw** increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. **getw** assumes no special alignment in the file.

Errors

If the *stream* is at **EOF**, the **EOF** indicator for the *stream* is set and **getc** returns **EOF**. If a read error occurs, the error indicator for the *stream* is set, **getc** returns **EOF** and sets **errno** to identify the error.

Under the following conditions, the functions **getc**, **getchar**, **fgetc** and **getw** fail and set **errno** to:

EAGAIN if the **O_NONBLOCK** flag is set for the underlying file descriptor and the process would have blocked in the read operation.

EBADF if the underlying file descriptor is not a valid file descriptor open for reading.

EINTR if a signal was caught during the **getc**, **getchar**, **fgetc** or **getw** call, and no data was transferred.

EIO if a physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

NOTICES

If the integer value returned by **getc**, **getchar**, or **fgetc** is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is implementation dependent.

The macro version of **getc** evaluates a *stream* argument more than once and may treat side effects incorrectly. In particular, **getc(*f++)** does not work sensibly. Use **fgetc** instead.

Because of possible differences in word length and byte ordering, files written using **putw** are implementation dependent, and may not be read using **getw** on a different processor.

Functions exist for all the above-defined macros. To get the function form, the macro name must be undefined (for example, **#undef getc**).

SEE ALSO

fclose(BA_OS), **ferror**(BA_OS), **fopen**(BA_OS), **fread**(BA_OS), **gets**(BA_LIB), **putc**(BA_LIB), **scanf**(BA_LIB), **stdio**(BA_LIB), **ungetc**(BA_LIB)

LEVEL

Level 1.

NAME

getdate – convert user format date and time

SYNOPSIS

```
#include <time.h>

struct tm *getdate(char *string);

extern int getdate_err;
```

DESCRIPTION

The routine `getdate()` converts user definable date and/or time specifications pointed to by *string*, into a `struct tm`. The structure declaration is in the `<time.h>` header file [see ctime(BA_LIB)].

User supplied templates are used to parse and interpret the input string. The templates are text files created by the user `DATEMSK`. The `DATEMSK` variable should be set to indicate the full pathname of the template file. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format. Upon successful completion, the function `getdate()` returns a pointer to a `struct tm`; otherwise, it returns `NULL` and the external variable `getdate_err` is set to indicate the error.

The following field descriptors are supported:

`%%`	same as `%`
`%a`	abbreviated weekday name
`%A`	full weekday name
`%b`	abbreviated month name
`%B`	full month name
`%c`	locale's appropriate date and time representation
`%d`	day of month (01 - 31; the leading 0 is optional)
`%e`	same as `%d`
`%D`	date as `%m/%d/%y`
`%h`	abbreviated month name
`%H`	hour (00 - 23)
`%I`	hour (01 - 12)
`%m`	month number (01 - 12)
`%M`	minute (00 - 59)
`%n`	same as `\n`
`%p`	locale's equivalent of either AM or PM
`%r`	time as `%I:%M:%S %p`
`%R`	time as `%H:%M`
`%S`	seconds (00 - 59)
`%t`	same as tab
`%T`	time as `%H:%M:%S`
`%w`	weekday number (Sunday = 0 - 6)
`%x`	locale's appropriate date representation
`%X`	locale's appropriate time representation
`%y`	year within century (00 - 99)
`%Y`	year as ccyy (*e.g.* 1986)

%Z time zone name or no characters if no time zone exists If the time zone supplied by %Z is not the same as the time zone getdate expects an invalid input specification error will result. Getdate calculates an expected time zone based on information supplied to the interface (such as the hour, day, and month).

The match between the template and input specification performed by getdate() is case insensitive.

The month and weekday names can consist of any combination of upper and lower case letters. The user can request that the input date or time specification be in a specific language by setting the LC_TIME category [see setlocale(BA_OS)].

Leading 0's are not necessary for the descriptors that allow leading 0's. However, at most two digits are allowed for those descriptors, including leading 0's. Extra whitespace in either the template file or in *string* is ignored.

The field descriptors %c, %x, and %X will not be supported if they include unsupported field descriptors.

The following example shows the possible contents of a template:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

The following are examples of valid input specifications for the above template:

```
getdate("10/1/87 4 PM");
getdate("Friday");
getdate("Friday September 18, 1987, 10:30:30");
getdate("24,9,1986 10:30");
getdate("at monday the 1st of december in 1986");
getdate("run job at 3 PM, december 2nd");
```

If the LC_TIME category is set to a German locale that includes freitag as a weekday name and oktober as a month name, the following would be valid:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

The following examples shows how local date and time specification can be defined in the template.

INVOCATION	LINE IN TEMPLATE
getdate("11/27/86")	%m/%d/%y
getdate("27.11.86")	%d.%m.%y
getdate("86-11-27")	%y-%m-%d
getdate("Friday 12:00:00")	%A %H:%M:%S

The following rules apply for converting the input specification into the internal format:

1 If only the weekday is given, today is assumed if the given day is equal to the current day and next week if it is less,

2 If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given (the first day of month is assumed if no day is given),

3 If no hour, minute and second are given the current hour, minute and second are assumed,

4 If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

The following examples help to illustrate the above rules assuming that the current date is `Mon Sep 22 12:19:47 EDT 1986` and the `LC_TIME` category is set to the default `"C"` locale.

INPUT	LINE IN TEMPLATE	DATE
`Mon`	`%a`	Mon Sep 22 12:19:47 EDT 1986
`Sun`	`%a`	Sun Sep 28 12:19:47 EDT 1986
`Fri`	`%a`	Fri Sep 26 12:19:47 EDT 1986
`September`	`%B`	Mon Sep 1 12:19:47 EDT 1986
`January`	`%B`	Thu Jan 1 12:19:47 EST 1987
`December`	`%B`	Mon Dec 1 12:19:47 EST 1986
`Sep Mon`	`%b %a`	Mon Sep 1 12:19:47 EDT 1986
`Jan Fri`	`%b %a`	Fri Jan 2 12:19:47 EST 1987
`Dec Mon`	`%b %a`	Mon Dec 1 12:19:47 EST 1986
`Jan Wed 1989`	`%b %a %Y`	Wed Jan 4 12:19:47 EST 1989
`Fri 9`	`%a %H`	Fri Sep 26 09:00:00 EDT 1986
`Feb 10:30`	`%b %H:%S`	Sun Feb 1 10:00:30 EST 1987
`10:30`	`%H:%M`	Tue Sep 23 10:30:00 EDT 1986
`13:30`	`%H:%M`	Mon Sep 22 13:30:00 EDT 1986

ERRORS

Upon failure, `NULL` is returned and the variable `getdate_err` is set to indicate the error.

The following is a complete list of the `getdate_err` settings and their corresponding descriptions.

1 the `DATEMSK` environment variable is null or undefined,
2 the template file cannot be opened for reading,
3 failed to get file status information,
4 the template file is not a regular file,
5 an error is encountered while reading the template file,
6 memory allocation failed (not enough memory available),
7 there is no line in the template that matches the input,

8 invalid input specification Example: February 31 or a time is specified that can not be represented in a time_t (representing the time in seconds since 00:00:00 UTC, January 1, 1970)

SEE ALSO

ctime(BA_LIB), ctype(BA_LIB), setlocale(BA_OS), strftime(BA_LIB), time(BA_OS).

LEVEL

Level 2, September 30, 1993. Replaced by `strptime(BA_LIB)`.

NAME

getenv – return value for environment name

SYNOPSIS

```
#include <unistd.h>
#include <stdlib.h>
char *getenv(const char *name);
```

DESCRIPTION

The function `getenv()` searches the environment for a string of the form *name*=*value* and returns a pointer to the *value* in the current environment if such a string is present. Otherwise, `NULL` is returned.

SEE ALSO

envvar(BA_ENV), exec(BA_OS), putenv(BA_LIB), system(BA_OS).

LEVEL

Level 1.

NAME

getgrent, **getgrgid**, **getgrnam**, **setgrent**, **endgrent**, **fgetgrent** – get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent (void);

struct group *getgrgid (gid_t gid);

struct group *getgrnam (const char *name);

void setgrent (void);

void endgrent (void);

struct group *fgetgrent (FILE *f);
```

DESCRIPTION

getgrent, **getgrgid**, and **getgrnam** each returns a pointer to a structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a ''group'' structure, defined in the **grp.h** header file with the following members:

```
char     *gr_name;    /* the name of the group */
gid_t     gr_gid;     /* the numerical group ID */
char    **gr_mem;    /* vector of pointers to member names */
```

When first called, **getgrent** returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. **getgrgid** searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found.

getgrnam searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a null pointer.

A call to **setgrent** has the effect of rewinding the group file to allow repeated searches. **endgrent** may be called to close the group file when processing is complete.

fgetgrent returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

Errors

getgrent, **getgrgid**, **getgrnam**, and **fgetgrent** return a null pointer on **EOF** or error. If a bad entry is encountered, **errno** is set to **EINVAL**. If the functions are unable to allocate sufficient space for the entry, **errno** is set to **ENOMEM**.

SEE ALSO

getlogin (BA_LIB), **getpwent** (BA_LIB),

NOTICES

All information is contained in a static area, so it must be copied if it is to be saved.

LEVEL

Level 2.

NAME

getlogin – get login name

SYNOPSIS

```
#include <stdlib.h>

char *getlogin(void);
```

DESCRIPTION

getlogin returns a pointer to the login name It may be used in conjunction with **getpwnam** to locate the correct password file entry when the same user id is shared by several login names.

If **getlogin** is called within a process that is not attached to a terminal, it returns a null pointer. The correct procedure for determining the login name is to call **cuserid**, or to call **getlogin** and if it fails to call **getpwuid**.

SEE ALSO

cuserid(BA_LIB), **getgrent**(BA_LIB), **getpwent**(BA_LIB)

LEVEL

Level 1.

NOTICES

The return values point to static data whose content is overwritten by each call.

NAME

getpass – read a password

SYNOPSIS

```
#include <unistd.h>

char *getpass(const char *prompt);
```

DESCRIPTION

getpass reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a null pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

Files

/dev/tty

NOTICES

The return value of **getpass** points to static data whose content is overwritten by each call.

Use the reentrant function **getpass_r** for multi-threaded applications.

LEVEL

Level 1.

NAME

getopt – get option letter from argument vector

SYNOPSIS

```
#include <stdio.h>

int getopt(int argc, char *const *argv, const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;
```

DESCRIPTION

The function `getopt()` is a command-line parser. It returns the next option letter in *argv* that matches a letter in *optstring*.

The function `getopt()` places in `optind` the *argv* index of the next argument to be processed. The external variable `optind` is initialized to `1` before the first call to the function `getopt()`.

The argument *optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may be separated from it by white space.

The variable `optarg` is set to point to the start of the option argument on return from `getopt()`.

When all options have been processed (i.e., up to the first non-option argument), the function `getopt()` returns `EOF`. The special option `--` may be used to delimit the end of the options; `EOF` will be returned and `--` will be skipped.

The following rules comprise the System V standard for command-line syntax:

RULE 1: Command names must be between two and nine characters.

RULE 2: Command names must include lower-case letters and digits only.

RULE 3: Option names must be a single character in length.

RULE 4: All options must be delimited by the - character.

RULE 5: Options with no arguments may be grouped behind one delimiter.

RULE 6: The first option-argument following an option may be preceded by white space.

RULE 7: Option arguments cannot be optional.

RULE 8: Groups of option arguments following an option must be separated by commas or separated by white space and quoted.

RULE 9: All options must precede operands on the command line.

RULE 10: The characters -- may be used to delimit the end of the options.

RULE 11: The order of options relative to one another should not matter.

RULE 12: The order of operands may matter and position-related interpretations should be determined on a command-specific basis.

RULE 13: The – character preceded and followed by white space should be used only to mean standard input.

RETURN VALUE

The function `getopt()` returns a question mark (`?`) when it encounters an option letter not included in *optstring*; it also prints an error message on `stderr` if `opterr` is set to non-`0` (`opterr` is initialized to 1). The value of the character that caused the error is in `optopt`. The message is printed in the standard error format. `getopt()` supports localized output messages. If the appropriate translated system messages are installed on the system, they are selected by the latest call to `setlocale()` (using the `LC_ALL` or `LC_MESSAGES` categories).

The label defined by a call to `setlabel()` will be used if available; otherwise, the name of the utility (`argv[0]`) will be used.

EXAMPLE

The following code fragment shows how one might process the options and arguments for a command that takes: mutually exclusive options `a` and `b`, exactly one of which is required; an optional option `i` which takes an option-argument; and at least two arguments.

```
main(int argc, char *argv[]
{
    int             opt, aflg=0, bflg=0, iflg=0, errflg=0, retval ;
    char            *cmdname, *ifile, *ofile ;
    FILE            *infile, *outfile ;
    extern int      optind, opterr, errno ;
    extern char     *optarg ;

    setlabel("UX:example");
    cmdname = argv[0] ;
    opterr = 0 ;    /* inhibit getopt err msg */
    while ( (opt=getopt(argc,argv,"abi:")) != EOF ) {
        switch ( opt ) {
        case 'a' :
            aflg += 1 ; break ;
        case 'b' :
            bflg += 1 ; break ;
        case 'i' :
            iflg += 1 ; ifile = optarg ; break ;
        default :   /* includes '?' case */
            errflg += 1 ; break ;
        }
    }
    if ( errflg>0 || aflg+bflg!=1 || iflg>1 || argc-optind<2 ) {
        usage_err_exit(cmdname) ;
    }
    if ( iflg == 0 ) {
        infile = stdin ;
    } else if ( (infile=fopen(ifile,"r")) == NULL ) {
        open_err_exit(cmdname,ifile,errno) ;
    }
```

(continues)

```
        for ( ; optind<argc ; optind+=1 ) {
            if ( (outfile=fopen(ofile=argv[optind],"r+")) == NULL ) {
                open_err_exit(cmdname,ofile,errno) ;
            }
            if ( (retval=do_work(aflg,bflg,infile,outfile)) != 0 ) {
                work_err_exit(cmdname,ofile,retval) ;
            }
            if ( fclose(outfile) != 0 ) {
                close_err_exit(cmdname,ofile,errno) ;
            }
        }
        exit(0) ;
    }
```

SEE ALSO

pfmt(BA_LIB) setlabel(BA_LIB)

LEVEL

Level 1.

NAME

getpwent, **getpwuid**, **getpwnam**, **setpwent**, **endpwent**, **fgetpwent** – manipulate password file entry

SYNOPSIS

```
#include <pwd.h>
#include <stdio.h>
struct passwd *getpwent (void);
struct passwd *getpwuid (uid_t uid);
struct passwd *getpwnam (const char *name);
void setpwent (void);
void endpwent (void);
struct passwd *fgetpwent (FILE *f);
```

DESCRIPTION

getpwent, **getpwuid**, and **getpwnam** each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a **passwd** structure, declared in the **pwd.h** header file:

```
struct passwd {
     char  *pw_name;
     char  *pw_passwd;
     uid_t pw_uid;
     gid_t pw_gid;
     char  *pw_dir;
     char  *pw_shell;
};
```

When first called, **getpwent** returns a pointer to the first **passwd** structure in the file; thereafter, it returns a pointer to the next **passwd** structure in the file. Thus successive calls can be used to search the entire file. **getpwuid** searches from the beginning of the file until a numerical user ID matching *uid* is found and returns a pointer to the particular structure in which it was found. **getpwnam** searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a null pointer.

A call to **setpwent** has the effect of rewinding the password file to allow repeated searches. **endpwent** may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next **passwd** structure in the stream *f*, which matches the format of **/etc/passwd**.

Files

/etc/passwd

Return Values

getpwent, **getpwuid**, **getpwnam**, and **fgetpwent** return a null pointer on **EOF** or error.

getpwent(BA_LIB)

SEE ALSO

getgrent(BA

NAME

gets, fgets – get a string from a stdio-stream

SYNOPSIS

```
#include <stdio.h>

char *gets(char *s);

char *fgets(char *s, int n, FILE *strm);
```

DESCRIPTION

The function `gets()` reads characters from the standard input stdio-stream, `stdin`, into the array pointed to by *s* until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

The function `fgets()` reads characters from *strm* into the array pointed to by *s* until *n*−1 characters are read, or a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

The functions `gets()` and `fgets()` may mark the `st_atime` field of the file associated with *strm* for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `getc()`, `getchar()`, `gets()` or `fscanf()` using *strm* that returns data not supplied by a prior call to `ungetc()`.

RETURN VALUE

If end-of-file is encountered and no characters have been read, *s* remains unchanged. If a read error occurs, the contents of *s* are undefined, the error indicator for the stdio-stream is set, and `NULL` is returned. Otherwise *s* is returned. If end-of-file is encountered, the end-of-file indicator for the stdio-stream is set.

ERRORS

Under the following conditions, the functions `gets()`, and `fgets()` fail and set `errno` to:

`EAGAIN` if the `O_NONBLOCK` flag is set for the underlying file descriptor and the process would have blocked in the read operation.

`EBADF` if the underlying file descriptor is not a valid file descriptor open for reading.

`EINTR` if a signal was caught during the `gets()`, or `fgets()` call and no data was transferred.

`EIO`

SEE ALSO

ferror(BA_OS), fopen(BA_OS), fread(BA_OS), getc(BA_LIB), puts(BA_LIB), scanf(BA_LIB).

LEVEL

Level 1.

NAME

getsubopt – parse sub options from a string.

SYNOPSIS

```
int getsubopt(char **optionp, char *tokens[], char **valuep);
```

DESCRIPTION

The function `getsubopt()` parses suboptions in a flag argument that were initially parsed by `getopt()` [see getopt(BA_LIB)]. These suboptions are separated by commas and may consist of either a single token, or a token-value pair separated by an equal sign. Because commas delimit suboptions in the option string, they are not allowed to be part of the suboption or the value of a suboption. Similarly, because the equal sign separates a token from its value, a token must not contain an equal sign. An example command that uses this syntax is `mount`. `mount` allows parameters to be specified with the `-o` switch as follows :

```
mount –o rw,hard,bg,wsize=1024  speed:/usr  /usr
```

In this example there are four suboptions: `rw`, `hard`, `bg`, and `wsize`, the last of which has an associated value of 1024.

`getsubopt()` takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or –1 if there was no match. If the option string at **optionp* contains only one suboption, `getsubopt()` updates **optionp* to point to the null at the end of the string, otherwise it isolates the suboption by replacing the comma separator with a null, and updates **optionp* to point to the start of the next suboption. If the suboption has an associated value, `getsubopt()` updates **valuep* to point to the value's first character. Otherwise it sets **valuep* to `NULL`.

The token vector is organized as a series of pointers to `NULL`-terminated strings. The end of the token vector is identified by `NULL`.

When `getsubopt()` returns, if **valuep* is not `NULL` then the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this subobtion is an error.

Additionally, when `getsubopt()` fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

EXAMPLE

The following code fragment shows how options may be processed to the `mount` command using `getsubopt()`.

```
char *myopts[] = {
#define READONLY   0
                   "ro",
#define READWRITE  1
                   "rw",
#define WRITESIZE  2
                   "wsize",
#define READSIZE   3
                   "rsize",
                   NULL};

main(argc, argv)
      int  argc;
      char **argv;
{
      int sc, c, errflag;
      char *options, *value;
      extern char *optarg;
      extern int optind;
      .
      .
      .
      while((c = getopt(argc, argv, "abf:o:")) != -1) {
            switch (c) {
            case 'a': /* process a option */
                  break;
            case 'b': /* process b option */
                  break;
            case 'f':
                  ofile = optarg;
                  break;
            case '?':
                  errflag++;
                  break;
```

(continues)

```
                case 'o':
                      options = optarg;
                      while (*options != '\0') {
                            switch(getsubopt(&options,myopts,&value) {
                            case READONLY : /* process ro option */
                                  break;
                            case READWRITE : /* process rw option */
                                  break;
                            case WRITESIZE : /* process wsize option */
                                  if (value == NULL) {
                                        error_no_arg();
                                        errflag++;
                                  } else
                                        write_size = atoi(value);
                                  break;
                            case READSIZE : /* process rsize option */
                                  if (value == NULL) {
                                        error_no_arg();
                                        errflag++;
                                  } else
                                        read_size = atoi(value);
                                  break;
                            default :
                                  /* process unknown token */
                                  error_bad_token(value);
                                  errflag++;
                                  break;
                            }
                      }
                      break;
                }
          }
          if (errflag) {
                /* print Usage instructions etc. */
          }
          for (; optind<argc; optind++) {
                /* process remaining arguments */
          }
          .
          .
          .
    }
```

SEE ALSO

getopt(BA_LIB).

LEVEL

Level 1.

NAME

gettxt – retrieve a text string

SYNOPSIS

`char *gettxt(char *`*msgid*`, char *`*dflt_str*`);`

DESCRIPTION

The routine `gettxt()` retrieves a text string from a message file. The arguments to the function are a mess' *msgid* and a default string *dflt_str* to be used if the retrieval fails.

The text strings are in files created by `mkmsgs` [see mkmsgs(AS_CMD)] and installed in

`/usr/lib/locale/`*locale*`/LC_MESSAGES`

directories.

The directory *locale* can be viewed as the language in which the text strings are written. The user can request that messages be displayed in a specific language by setting the environment variable `LC_MESSAGES`. If `LC_MESSAGES` is not set the environment variable `LANG` will be used.

If `LANG` is not set, the locale in which the strings will be retrieved is the C locale and the files containing the strings are in

`/usr/lib/locale/C/LC_MESSAGES/*`.

The user can also change the language in which the messages are displayed by invoking the `setlocale()` [see setlocale(BA_OS)] function with the appropriate arguments. If the locale is explicitly changed (via `setlocale()`), the pointers returned by `gettxt()` may no longer be valid.

The following depicts the acceptable syntax of *msgid* for a call to `gettxt()`:

msgfilename`:`*msgnumber*

The argument *msgid* consists of two fields separated by a colon. The first field is used to indicate the file that contains the text strings and is limited to 14 characters. These characters must be selected from a set of all character values excluding `\0` (null) and the ASCII code for `/` (slash) and `:` (colon). The names of message files must be the same as the names of files created by `mkmsgs()` and installed in `/usr/lib/locale/`*locale*`/LC_MESSAGES/*`. If no file name is specified, `gettxt()` will use the name specified with `setcat()`. [see setcat(BA_LIB)] The numeric field indicates the sequence number of the string in the file. The strings are numbered from `1`.

If *msgfilename* does not exist in the locale (specified by the last call to `setlocale` using the `LC_ALL` or `LC_MESSAGES` categories), or if the message number is out of bounds, `gettxt` attempts to retrieve the message from the C locale. If this second retrieval fails, `gettxt` uses *dflt_str*.

If *msgfilename* is omitted, `gettxt` attempts to retrieve the string from the default catalog specified by the last call to `setcat`.

`gettxt` outputs `Message not found!!\n` if:

- *msgfilename* is not a valid catalog name as defined above
- no catalog is specified (either explicitly or via `setcat`)
- *msgnumber* is not a positive number
- no message could be retrieved and *dflt_str* was omitted

FILES

`/usr/lib/locale/C/LC_MESSAGES/*`	Default message files created by `mkmsgs()`
`/usr/lib/locale/`*locale*`/LC_MESSAGES/*`	message files for different languages created by `mkmsgs()`

EXAMPLE

In the following code fragment:

```
gettxt("test:10", "hello world\n")
gettxt("test:10", "")
setcat("test");
gettxt(":10", "hello world\n")
```

`test` is the name of the file that contains the messages; `10` is the message number.

SEE ALSO

envvar(BA_ENV), gettxt(BU_CMD), mkmsgs(AS_CMD), setcat(BA_LIB), setlocale(BA_OS), srchtxt(AS_CMD).

LEVEL

Level 1.

NAME

getwc, **getwchar**, **fgetwc** – get next wide character from a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t getwc(FILE *stream);

wint_t getwchar(void);

wint_t fgetwc(FILE *stream);
```

DESCRIPTION

fgetwc transforms the next multibyte character from the named input stream into a wide character, and returns it. It also increments the file pointer, if defined, by one multibyte character. **getwchar** is defined as **getwc(stdin)**.

getwc behaves like **fgetwc**, except that **getwc** may be implemented as a macro which evaluates *stream* more than once.

Errors

These functions return the constant **WEOF** and sets the stream's end-of-file indicator at the end-of-file. They return **WEOF** if an error is found. If the error is an I/O error, the error indicator is set. If it is due to an invalid or incomplete multibyte character, **errno** is set to **EILSEQ**.

NOTICES

If the value returned by **getwc**, **getwchar**, or **fgetwc** is compared with the integer constant **WEOF** after being stored in a **wchar_t** object, the comparison may not succeed.

SEE ALSO

fclose(BA_OS), **ferror**(BA_OS), **fopen**(BA_OS), **putwc**(BA_LIB), **scanf**(BA_LIB), **stdio**(BA_LIB),

LEVEL

Level 1.

NAME

fgetws – get a **wchar_t** string from a stream

SYNOPSIS

```
#include <stdio.h>
#include <widec.h>

wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
```

DESCRIPTION

fgetws reads wide characters from the stream, converts them to **wchar_t** characters, and places them in the **wchar_t** array pointed to by *s*. **fgetws** reads until *n*–**1** **wchar_t** characters are transferred to *s*, or a newline character or an end-of-file condition is encountered. The **wchar_t** string is then terminated with a **wchar_t** null character.

Errors

If end-of-file or a read error is encountered and no characters have been transformed, no **wchar_t** characters are transferred to *s* and a null pointer is returned and the error indicator for the stream is set. If the read error is an illegal byte sequence, **errno** is set to **EILSEQ**. If end-of-file is encountered, the **EOF** indicator for the stream is set. Otherwise, *s* is returned.

SEE ALSO

fread(BA_OS), **getwc**(BA_LIB), **scanf**(BA_LIB), **stdio**(BA_LIB)

LEVEL

Level 1.

NAME

glob, globfree – generate pathnames matching a pattern

SYNOPSIS

```
#include <glob.h>

int glob(const char *pattern, int flags,
   int (*errfunc)(const char *epath, int eerrno), glob_t *pglob);

void globfree(glob_t *pglob);
```

DESCRIPTION

These functions are part of the X/Open Portability Guide Issue 4 optional POSIX2 C-Language Binding feature group.

Return Values

glob returns **GLOB_NOSYS** and sets **errno** to **ENOSYS**.

globfree returns and sets **errno** to **ENOSYS**.

USAGE

Administrator.

SEE ALSO

fnmatch(BA_LIB), **wordexp**(BA_LIB)

LEVEL

Level 1.

NAME

grantpt – grant access to the slave pseudo-terminal device

SYNOPSIS

```
int grantpt(int fildes);
```

DESCRIPTION

The function `grantpt()` changes the mode and ownership of the slave pseudo-terminal device associated with its master pseudo-terminal counter part. *fildes* is the file descriptor returned from a successful open of the master pseudo-terminal device. A `setuid()` root program [see setuid(BA_OS)] is invoked to change the permissions. The user ID of the slave is set to the real UID of the calling process and the group ID is set to a reserved group. The permission mode of the slave pseudo-terminal is set to readable, writeable, by the owner and writeable by the group.

RETURN VALUE

Upon successful completion, the function `grantpt()` returns a value of `0`; otherwise, it returns a value of `-1`. Failure could occur if *fildes* is not an open file descriptor, is not associated with a master pseudo-terminal device, or if the corresponding slave device could not be accessed.

SEE ALSO

open(BA_OS), ptsname(BA_LIB), setuid(BA_OS), unlockpt(BA_LIB).

LEVEL

Level 1.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch(ENTRY item, ACTION action);

int hcreate(unsigned nel);

void hdestroy(void);
```

DESCRIPTION

The function `hsearch()` is a hash-table search routine. It returns a pointer into a hash table indicating the location at which an entry can be found. The comparison function used by `hsearch()` is the function `strcmp()` [see string(BA_LIB)].

The argument *item* is a structure of type `ENTRY` (defined in search.h header[see search(BA_ENV)]) containing two pointers: *item.key* pointing to the comparison key and *item.data* pointing to any other data to be associated with that key. (Pointers to types other than `void` should be cast to pointer-to-void.)

The argument *action* is a member of an enumeration type `ACTION`, indicating the disposition of the entry if it cannot be found in the table.

`ENTER` indicates that the *item* should be inserted in the table at an appropriate point. Given a duplicate of an existing item, the new *item* is not entered, and `hsearch()` returns a pointer to the existing item.

`FIND` indicates that no entry should be made. Unsuccessful resolution is indicated by the return of `NULL`.

The function `hcreate()` allocates sufficient space for the table and must be called before `hsearch()` is used. The value of *nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The function `hdestroy()` destroys the search table and may be followed by another call to `hcreate()`.

RETURN VALUE

The function `hsearch()` returns `NULL` if either the *action* is `FIND` and the *item* could not be found or the *action* is `ENTER` and the table is full.

The function `hcreate()` returns `0` if it cannot allocate sufficient space for the table.

EXAMPLE

The example reads in strings followed by two numbers and stores them in a hash table. It then reads in strings and finds the entry in the table and prints it.

```
#include <stdio.h>
#include <search.h>
#include <string.h>

struct info {          /* these are in the table */
    int age, room;     /* apart from the key. */
};
#define NUM_EMPL 5000  /* # of elements in the table */

main ()
{
    char string_space[NUM_EMPL*20];    /* space for strings */
    struct info info_space[NUM_EMPL];  /* space for employee info */
    char *str_ptr = string_space;      /* next avail space for strings */
    struct info *info_ptr = info_space;      /* next avail space for info */
    ENTRY item, *found_item;
    char name_to_find[30];      /* name to look for in table */
    int i = 0;

    /* create table */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
        &info_ptr->room) != EOF && i++ < NUM_EMPL) {
        /* put info in structure, and structure in item */
        item.key = str_ptr;
        item.data = (void *)info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;
        (void) hsearch(item, ENTER);   /* put item into table */
    }
    /* access table */
    item.key = name_to_find;
    while (scanf("%s", item.key) != EOF) {
        if ((found_item = hsearch(item, FIND)) != NULL) {
        /* if item is in the table */
        (void) printf("found %s, age = %d, room = %d\n",
            found_item->key,
            ((struct info *)found_item->data)->age,
            ((struct info *)found_item->data)->room);
        } else {
        (void) printf("no such employee %s\n",
            name_to_find);
        }
    }
}
```

SEE ALSO

bsearch(BA_LIB), lsearch(BA_LIB), malloc(BA_OS), string(BA_LIB), tsearch(BA_LIB).

FUTURE DIRECTIONS

The restriction of having only one hash search table active at any given time will be removed.

LEVEL

Level 1.

NAME

hyperbolic: sinh, cosh, tanh, asinh, acosh, atanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(double x);

double cosh(double x);

double tanh(double x);

double asinh(double x);

double acosh(double x);

double atanh(double x);
```

DESCRIPTION

The functions `sinh()`, `cosh()`, and `tanh()` return, respectively, the hyperbolic sine, cosine, and tangent of their argument.

The functions `asinh()`, `acosh()`, and `atanh()` return, respectively, the inverse hyperbolic sine, cosine, and tangent of their argument.

RETURN VALUE

A macro `HUGE_VAL` will be defined by the `<math.h>` header file. This macro evaluates to a positive double expression, not necessarily representable as a float. On implementations that support the IEEE 754 standard, `HUGE_VAL` evalutates to $+\infty$.

The functions `sinh()` and `cosh()` will return `HUGE_VAL` (`sinh()` will return –`HUGE_VAL` for negative *x*) and set errno to `ERANGE` when the correct value overflows.

The function `acosh()` returns an implementation-defined value (IEEE `NaN` or equivalent if available) and sets `errno` to `EDOM` when its argument is less than 1.0.

The function `atanh()` returns an implementation-defined value (IEEE `NaN` or equivalent if available) and sets `errno` to `EDOM` when its argument has absolute value greater than 1.0.

On implementations which support IEEE `NaN`, if an input parameter is `NaN`, then the function will return `NaN` and set errno to `EDOM`.

LEVEL

Level 1.

NAME

hypot – Euclidean distance function

SYNOPSIS

```
#include <math.h>

hypot(double x, double y);
```

DESCRIPTION

The function `hypot()` returns $\sqrt{x^2+y^2}$ taking precautions against unwarranted overflows.

RETURN VALUE

A macro `HUGE_VAL` will be defined by the `<math.h>` header file. This macro evaluates to a positive double expression, not necessarily representable as a float. On implementations that support the IEEE 754 standard, `HUGE_VAL` evalutates to $+\infty$.

On implementations which support IEEE `NaN`, if an input parameter is `NaN`, then the function will return `NaN` and set errno to `EDOM`.

The only exception is that if one of the arguments is `NaN` and the other argument is $\pm\infty$, `HUGE_VAL` is returned with no error indication.

The function `hypot()` will return `HUGE_VAL` and set `errno` to `ERANGE` when the correct value overflows.

LEVEL

Level 1.

NAME

iconv_close – code conversion deallocation function

SYNOPSIS

```
#include <iconv.h>

int iconv_close(iconv_t cd);
```

DESCRIPTION

iconv_close deallocates the conversion descriptor *cd*, and all data contained within it. If a file descriptor or similar facility is used within the descriptor, it is closed and deallocated.

Return Values

If *iconv_close* encounters no errors, it returns zero. Otherwise **-1** is returned, and **errno** is set.

Errors

EBADF *cd* may be an invalid conversion descriptor.

USAGE

Administrator.

SEE ALSO

iconv(AU_CMD), **iconv_open**(BA_LIB),

LEVEL

Level 1.

NAME

iconv_open – code conversion allocation function.

SYNOPSIS

```
#include <iconv.h>

iconv_t iconv_open(const char *tocode, const char *fromcode);
```

DESCRIPTION

iconv_open returns a conversion descriptor for the codeset conversion from codeset *fromcode* to codeset *tocode*. This descriptor is used on subsequent calls to *iconv*.

The allowable values for *fromcode* and *tocode* are dependent on the implementation. This is also true for the different combinations allowed.

A conversion descriptor is valid until the creating process terminates, or until it is passed to **iconv_close**.

Return Values

If **iconv_open** completes successfully, a conversion descriptor is returned. Should the function fail,**iconv_open** returns **(iconv_t)-1** and **errno** is set to indicate an error.

Errors

EMFILE	There may be no more file descriptors free for the process.
ENFILE	There may be too many open files on the system.
ENOMEM	Not enough memory.
EINVAL	The implementation does not support the specified conversion.

USAGE

Administrator.

SEE ALSO

iconv(BU_CMD), **iconv**(BA_LIB), **iconv_close**(BA_LIB), **iconvh**(BA_LIB)

LEVEL

Level 1.

NOTICES

In some implementations, this function uses dynamic memory allocation (**malloc**) to provide space for internal buffer areas. If there is not enough space to cater for these buffers, it is likely that the **iconv_open** function will fail.

Applications that are portable must assume that conversion descriptors are invalidated after one of the *exec* functions is called.

NAME

initgroups – initialize the supplementary group access list

SYNOPSIS

```
#include <sys/types.h>

int initgroups(const char *name, gid_t basegid);
```

DESCRIPTION

The function `initgroups()` gets the supplementary group membership for the user specified by *name* and then initializes the supplementary group access list of the calling process using `setgroups()` [see `setgroups()` in getgroups(BA_OS)]. The *basegid* group ID is also included in the supplementary group access list. This is typically the real group ID from the password file.

If the number of groups, including the *basegid* entry, exceeds `{NGROUPS_MAX}`, then subsequent group entries are ignored.

RETURN VALUE

Upon successful completion, the function `initgroups()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following condition, the function `initgroups()` fails and sets `errno` to:

`EPERM` if the calling process does not have appropriate privileges.

SEE ALSO

getgroups(BA_OS), group(BA_ENV).

LEVEL

Level 1.

NAME

isastream – test a file descriptor

SYNOPSIS

```
int isastream(int fildes);
```

DESCRIPTION

The function `isastream()` determines if a file descriptor represents a STREAMS file. *fildes* refers to an open file.

RETURN VALUE

Upon successful completion, the function `isastream()` returns a value of `1` if *fildes* represents a STREAMS file and `0` if not. Otherwise, the function `isastream()` returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `isastream()` fails and sets `errno` to:

`EBADF` if *fildes* is not a valid open file.

SEE ALSO

streams(BA_DEV).

LEVEL

Level 1.

NAME

isnan, isnand – test for NaN

SYNOPSIS

```
#include <math.h>

int isnan(double x);

int isnand (double x);
```

DESCRIPTION

The function `isnan()` tests whether *x* is IEEE `NaN`. The functionality of `isnand()` is identical to that of `isnan()`.

RETURN VALUE

The functions `isnan()` and `isnand()` return non-zero if *x* is IEEE `NaN`; otherwise it returns `0`.

The function `isnan()` always returns `0` on implementations that do not support IEEE `NaN`.

SEE ALSO

math(BA_ENV).

LEVEL

Level 1.

The following interface definition has been moved to Level 2 effective April 1991.

```
int isnand (double x);
```

NAME

lfmt – **lfmt**, **vlfmt**; display error message in standard format and pass to logging and monitoring services

SYNOPSIS

```
#include <pfmt.h>

int lfmt(FILE *stream, long flags, char *format, ... /* arg */);

#include <stdarg.h>
#include <pfmt.h>
int vlfmt(FILE *stream, long flags, char *format, va_list ap);
```

DESCRIPTION

lfmt retrieves a format string from a locale-specific message database (unless **MM_NOGET** is specified) and uses it for **printf** style formatting of *args*. The output is displayed on *stream*. If *stream* is **NULL**, no output is displayed. **lfmt** encapsulates the output in the standard error message format (unless **MM_NOSTD** is specified, in which case the output is simply **printf**-like).

lfmt forwards its output to the logging and monitoring facility, even if *stream* is null. Optionally, **lfmt** will display the output on the console, with a date and time stamp.

If the **printf** format string is to be retrieved from a message database, the *format* argument must have the following structure:

catalog:*msgnum*:*defmsg*.

If **MM_NOGET** is specified, only the *defmsg* part must be specified.

catalog indicates the message database that contains the localized version of the format string. *catalog* is limited to 14 characters. These characters must be selected from a set of all character values, excluding **\0** (null) and the ASCII codes for **/** (slash) and **:** (colon).

msgnum must be a positive number that indicates the index of the string into the message database.

If *catalog* does not exist in the locale (specified by the last call to **setlocale** using the **LC_ALL** or **LC_MESSAGES** categories), or if the message number is out of bounds, **lfmt** attempts to retrieve the message from the C locale. If this second retrieval fails, **lfmt** uses the *defmsg* part of the *format* argument.

If *catalog* is omitted, **lfmt** attempts to retrieve the string from the default catalog specified by the last call to **setcat**. In this case, the *format* argument has the following structure:

:*msgnum*:*defmsg*.

lfmt outputs **Message not found!!\n** as the format string if:

– *catalog* is not a valid catalog name as defined above

- no catalog is specified (either explicitly or via **setcat**)
- *msgnum* is not a positive number, or if no message could be retrieved from the message databases and *defmsg* was omitted.

The *flags* determine the type of output (i.e., whether the *format* should be interpreted as is or encapsulated in the standard message format), and the access to message catalogs to retrieve a localized version of *format*. The *flags* are composed of several groups, and can take the following values (one from each group):

Output format control

MM_NOSTD do not use the standard message format, interpret *format* as a **printf** *format*. Only *catalog access control flags*, *console display control*, and *logging information* should be specified if **MM_NOSTD** is used; all other flags will be ignored.

MM_STD output using the standard message format (default, value 0).

Catalog access control

MM_NOGET do not retrieve a localized version of *format*. In this case, only the *defmsg* part of the *format* is specified.

MM_GET retrieve a localized version of *format*, from the *catalog*, using *msgnum* as the index and *defmsg* as the default message (default, value 0).

Severity (standard message format only)

MM_HALT generates a localized version of **HALT**.

MM_ERROR generates a localized version of **ERROR** (default, value 0).

MM_WARNING generates a localized version of **WARNING**.

MM_INFO generates a localized version of **INFO**.

Additional severities can be defined. Add-on severities can be defined with number-string pairs with numeric values from the range [5,255], using **addsev()**. The numeric value ORed with other *flags* will generate the specified severity.

If the severity is not defined, **lfmt** uses the string **SEV=***N* where *N* is replaced by the integer severity value passed in *flags*.

Multiple severities passed in *flags* will not be detected as an error. Any combination of severities will be summed and the numeric value will cause the display of either a severity string (if defined) or the string **SEV=***N* (if undefined).

Action

MM_ACTION specifies an action message. Any severity value is superseded and replaced by a localized version of **TO FIX**.

Console display control

MM_CONSOLE display the message to the console in addition to the specified *stream*.

MM_NOCONSOLE do not display the message to the console in addition to the specified *stream* (default, value 0).

Logging information

Major classification
identifies the source of the condition. Identifiers are: **MM_HARD** (hardware), **MM_SOFT** (software), and **MM_FIRM** (firmware).

Message source subclassification
identifies the type of software in which the problem is spotted. Identifiers are: **MM_APPL** (application), **MM_UTIL** (utility), and **MM_OPSYS** (operating system).

Standard Error Message Format

lfmt displays error messages in the following format:

label: *severity*: *text*

If no *label* was defined by a call to **setlabel**, the message is displayed in the format:

severity: *text*

If **lfmt** is called twice to display an error message and a helpful *action* or recovery message, the output can look like:

label: *severity*: *text*
label: **TO FIX:** *text*

vlfmt is the same as **lfmt** except that instead of being called with a variable number of arguments, it is called with an argument list as defined by the **stdarg.h** header file.

The **stdarg.h** header file defines the type **va_list** and a set of macros for advancing through a list of arguments whose number and types may vary. The argument *ap* to **vlfmt** is of type **va_list**. This argument is used with the **stdarg.h** header file macros **va_start**, **va_arg** and **va_end** [see **va_start**, **va_arg**, and **va_end** in **stdarg**(5)]. The EXAMPLE sections below show their use.

The macro **va_alist** is used as the parameter list in a function definition as in the function called **error** in the example below. The macro **va_start(***ap***,)**, where *ap* is of type **va_list**, must be called before any attempt to traverse and access unnamed arguments. Calls to **va_arg(***ap***,** *atype***)** traverse the argument list. Each execution of **va_arg** expands to an expression with the value and type of the next argument in the list *ap*, which is the same object initialized by **va_start**. The argument *atype* is the type that the returned argument is expected to be. The **va_end(***ap***)** macro must be invoked when all desired arguments have been accessed. [The argument list in *ap* can be traversed again if **va_start** is called again after **va_end**.] In the example below, **va_arg** is executed first to retrieve the format string passed to **error**. The remaining **error** arguments, *arg1*, *arg2*, ..., are given to **vlfmt** in the argument *ap*.

RETURN VALUE

On success, **lfmt** and **vlfmt** return the number of bytes transmitted. On failure, they return a negative value:

–1 write error to *stream*

–2 cannot log and/or display at console.

EXAMPLE

lfmt example 1

```
setlabel("UX:test");
lfmt(stderr, MM_ERROR|MM_CONSOLE|MM_SOFT|MM_UTIL,
          "test:2:Cannot open file: %s\n", strerror(errno));
```

displays the message to *stderr* and to the console and makes it available for logging:

```
UX:test: ERROR: Cannot open file: No such file or directory
```

lfmt example 2

```
setlabel("UX:test");
lfmt(stderr, MM_INFO|MM_SOFT|MM_UTIL,
       "test:23:test facility is enabled\n");
```

displays the message to *stderr* and makes it available for logging:

```
UX:test: INFO: test facility enabled
```

vlfmt example

The following demonstrates how **vlfmt** could be used to write an **errlog** routine:

```
#include <pfmt.h>
#include <stdarg.h>
. . .
/*
 *   errlog should be called like
 *         errlog(log_info, format, arg1, ...);
 */
void errlog(long log_info, const char *format, ...)

{
    va_list ap;

    va_start(ap,format);
    (void) vlfmt(stderr, log_info|MM_ERROR, format, ap);
    va_end(ap);
    (void) abort();
}
```

SEE ALSO

addsev(BA_LIB), envvar(BA_ENV), gettxt(BA_LIB), pfmt(BA_LIB), lfmt(BU_CMD), pfmt(BU_CMD), printf(BA_LIB), setcat(BA_LIB), setlabel(BA_LIB), setlocale(BA_LIB).

LEVEL

Level 2, April 1991.

NAME

lgamma, gamma – log gamma functions

SYNOPSIS

```
#include <math.h>
double lgamma(double x);
†double gamma(double x);
extern int signgam;
```

DESCRIPTION

The functions `lgamma()` and `gamma()` return $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as:

$$\int_0^\infty e^{-t} t^{x-1} dt$$

The sign of $\Gamma(x)$ is returned in the external integer `signgam`. If x is negative then it must not have an integral value. x may not be zero.

The following code fragment might be used to calculate Γ:

```
if ((y = lgamma(x)) > LN_MAXDOUBLE)
    error();
y = signgam * exp(y);
```

RETURN VALUE

On implementations that support IEEE NaN, if an input parameter is NaN, then the function will return NaN and set errno to EDOM.

A macro HUGE_VAL will be defined by the <math.h> header file. This macro evaluates to a positive double expression, not necessarily representable as a float. On implementations that support the IEEE 754 standard, HUGE_VAL evaluates to $+\infty$.

For non-positive integer arguments, `gamma()` and `lgamma()` return HUGE_VAL and set `errno` to EDOM.

If the correct value would overflow, `gamma()` and `lgamma()` return HUGE_VAL and set `errno` to ERANGE.

SEE ALSO

exp(BA_LIB)

FUTURE DIRECTIONS

On a system that supports the IEEE 754 standard, if the value of x for `lgamma()` is -infinity, `lgamma` will return IEEE NaN and set errno to EDOM.

The function `gamma()` will be removed from a future issue of the SVID.

LEVEL

Level 2.

`gamma` is Level 2, effective September 30, 1993.

NAME

localeconv – set the components of a locale

SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv(void);
```

DESCRIPTION

The function `localeconv()` sets the components of an object with type `struct lconv` with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale [see setlocale(BA_OS)]. `struct lconv` includes the following members:

```
char *decimal_point;
char *thousands_sep;
char *grouping;
char *int_curr_symbol;
char *currency_symbol;
char *mon_decimal_point;
char *mon_thousands_sep;
char *mon_grouping;
char *positive_sign;
char *negative_sign;
char int_frac_digits;
char frac_digits;
char p_cs_precedes;
char p_sep_by_space;
char n_cs_precedes;
char n_sep_by_space;
char p_sign_posn;
char n_sign_posn;
```

The members of the structure with type `char *` are strings, any of which (except `decimal_point`) can point to `""`, to indicate that the value is not available in the current locale or is of zero length. The members with type `char` are nonnegative numbers, any of which can be `CHAR_MAX` (defined in <`limits.h`>) to indicate that the value is not available in the current locale. The members are the following:

`char *decimal_point`
: The decimal-point character used to format non-monetary quantities.

`char *thousands_sep`
: The character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.

`char *grouping`
: A string in which each element is taken as an integer that indicates the number of digits that comprise the current group in a formatted non-monetary quantity. The elements of `grouping` are interpreted according to the following:

`CHAR_MAX` No further grouping is to be performed.

`0` The previous element is to be repeatedly used for the remainder of the digits.

other The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

`char *int_curr_symbol`
The international currency symbol applicable to the current locale, left-justified within a four-character space-padded field. The character sequences should match with those specified in: *ISO 4217 Codes for the Representation of Currency and Funds*.

`char *currency_symbol`
The local currency symbol applicable to the current locale.

`char *mon_decimal_point`
The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`
The separator for groups of digits to the left of the decimal-point in formatted monetary quantities.

`char *mon_grouping`
A string in which each element is taken as an integer that indicates the number of digits that comprise the current group in a formatted monetary quantity. The elements of `mon_grouping` are interpreted according to the rules described under `grouping`.

`char *positive_sign`
The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`
The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`
The number of fractional digits (those to the right of the decimal point) to be displayed in an internationally formatted monetary quantity.

`char frac_digits`
The number of fractional digits (those to the right of the decimal-point) to be displayed in a formatted monetary quantity.

`char p_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

`char p_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

`char n_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative formatted monetary quantity.

`char n_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

`char p_sign_posn`
Set to a value indicating the positioning of the `positive_sign` for a non-negative formatted monetary quantity. The value of `p_sign_posn` is interpreted according to the following:

`0` Parentheses surround the quantity and `currency_symbol`.

`1` The sign string precedes the quantity and `currency_symbol`.

`2` The sign string succeeds the quantity and `currency_symbol`.

`3` The sign string immediately precedes the `currency_symbol`.

`4` The sign string immediately succeeds the `currency_symbol`.

`char n_sign_posn`
Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity. The value of `n_sign_posn` is interpreted according to the rules described under `p_sign_posn`.

RETURN VALUE

The function `localeconv()` returns a pointer to the filled-in object. The structure pointed to by the return value may be overwritten by a subsequent call to `localeconv()`.

EXAMPLE

The following table illustrates the rules used by four countries to format monetary quantities.

Country	Positive format	Negative format	International format
Italy	L.1.234	-L.1.234	ITL.1.234
Netherlands	F 1.234,56	F -1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by `localeconv()` are as follows:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0

```
p_sign_posn         1        1             1           1
n_sign_posn         1        4             2           2
```

Note that the `mon_grouping` value (`"\3"` for all the above countries) is the ANSI C encoding for a string literal whose value is octal 3 (null-terminated). Hence, grouping is by threes (repeating) because the string is interpreted as an integer value of 3 followed by zero.

SEE ALSO

setlocale(BA_OS).

LEVEL

Level 1.

NAME

lsearch, lfind – linear search and update

SYNOPSIS

```
#include <sys/types.h>
#include <search.h>

void *lsearch(const void *key, void *base,
     size_t *nelp, size_t width,
     int (*compar)(const void *, const void *));

void *lfind(const void *key, const void *base,
     size_t *nelp, size_t width,
     int (*compar)(const void *, const void *));
```

DESCRIPTION

The function `lsearch()` is a linear search routine. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. The value of *key* points to the datum to be sought in the table. The value of *base* points to the first element in the table. The value of *nelp* points to an integer containing the current number of elements in the table. The value of *width* is the size of an element in bytes. The variable pointed to by *nelp* is incremented if the datum is added to the table. The value of *compar* is the name of the comparison function which the user must supply (`strcmp()`, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

The function `lfind()` is the same as `lsearch()` except that if the datum is not found, it is not added to the table. Instead, a `null` pointer is returned.

RETURN VALUE

If the datum is found, both the functions `lsearch()` and `lfind()` return a pointer to it. Otherwise, the function `lfind()` returns `NULL` and the function `lsearch()` returns a pointer to the newly added element.

USAGE

The pointers to the key and the element at the base of the table may be pointers to any type.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The value required should be cast into type pointer-to-element.

Space for the table must be managed by the application-program. Undefined results can occur if there is not enough room in the table to add a new item.

EXAMPLE

The following code fragment will read in ≤ `TABSIZE` strings of length ≤ `ELSIZE` and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>
#include <string.h>

#define TABSIZE 50
#define ELSIZE 120
```

```
char line[ELSIZE], tab[TABSIZE][ELSIZE];
size_t nel = 0;
. . .
while (fgets(line, ELSIZE, stdin) != NULL &&
    nel < TABSIZE)
        (void) lsearch((void *) line, (void *) tab,
                       &nel, ELSIZE, strcmp);
. . .
```

SEE ALSO

bsearch(BA_LIB), hsearch(BA_LIB), tsearch(BA_LIB).

FUTURE DIRECTIONS

`NULL` will be returned by the function `lsearch()`, with `errno` set appropriately, if there is not enough room in the table to add a new item.

LEVEL

Level 1.

NAME

makecontext, **swapcontext** – manipulate user contexts

SYNOPSIS

```
#include <ucontext.h>

void makecontext (ucontext_t *ucp, (void *func)(), int argc, . . . );

int swapcontext (ucontext_t *oucp, ucontext_t *ucp);
```

DESCRIPTION

These functions are useful for implementing user-level context switching between multiple threads of control within a process.

makecontext modifies the context specified by *ucp*, which has been initialized using **getcontext**; when this context is resumed using **swapcontext** or **setcontext** [see **getcontext**(BA_OS)], program execution continues by calling the function *func*, passing it the arguments that follow *argc* in the **makecontext** call. Before a call is made to **makecontext**, the context being modified should have a stack allocated for it. The value of *argc* must match the number of integers passed to *func*, otherwise the behavior is undefined.

The **uc_link** field is used to determine the context that will be resumed when the context being modified by **makecontext** returns. The **uc_link** field should be initialized prior to the call to **makecontext**.

swapcontext saves the current context in the context structure pointed to by *oucp* and sets the context to the context structure pointed to by *ucp*.

These functions will fail if the following is true:

ENOMEM *ucp* does not have enough stack left to complete the operation.

SEE ALSO

exit(BA_OS), **getcontext**(BA_OS), **sigaction**(BA_OS), **sigprocmask**(BA_OS), **ucontext**(BA_ENV)

RETURN VALUE

On successful completion, **swapcontext** return a value of zero. Otherwise, a value of –1 is returned and **errno** is set to indicate the error.

LEVEL

Level 1.

NAME

mbchar: **mbtowc**, **wctomb**, **mblen**, **mbrtowc**, **wcrtomb**, **mbrlen** – multibyte character handling

SYNOPSIS

```
#include <stdlib.h>

int mbtowc(wchar_t *pwc, const char *s, size_t n);

int wctomb(char *s, wchar_t wchar);

int mblen(const char *s, size_t n);

#include <wchar.h>

int mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);

int wcrtomb(char *s, wchar_t wc, mbstate_t *ps);

int mbrlen(const char *s, size_t n, mbstate_t *ps);
```

DESCRIPTION

Traditional computer systems used to assume that a character of a natural language could be represented in one byte of storage. Languages such as Japanese, Korean, Chinese, or Taiwanese, however, require more than one byte of storage to represent a character. These characters are called ''multibyte characters''. Such character sets are often called ''extended character sets''.

The number of bytes of storage required by a character in a given locale is defined in the **LC_CTYPE** category of the locale [see **setlocale**(BA_OS)]. The maximum

If *s* is a null pointer, **mbrtowc** and **wcrtomb** determine the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). The resulting state described is the initial conversion state. In this case, the value of the *pwc* is ignored.

If *s* is not a null pointer, **mbrtowc** determines the number of bytes that are contained in the multibyte character (plus any leading shift sequences) pointed to by *s*, produces the value of the corresponding wide character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc*. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

If *s* is not a null pointer, **wcrtomb** determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by *wc* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most **MB_CUR_MAX** bytes are stored. If *wc* is a null wide character, the resulting state described is the initial conversion state.

mbrlen is equivalent to the following call:

```
m
```

–2 if the next *n* bytes form an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed; this situation does not apply since the multibyte encoding is stateless.

–1 if an encoding error occurs (when the next *n* or fewer bytes do not form a complete and valid multibyte character); the value of the macro **EILSEQ** is stored in **errno**, but the conversion state is unchanged.

SEE ALSO

stdlib(BA_ENV), **mbstring**(BA_LIB), **setlocale**(BA_OS),

LEVEL

Level 1.

NAME

mbsinit – test for initial multibyte conversion state

SYNOPSIS

```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

DESCRIPTION

If *ps* is not a null pointer, **mbsinit** determines whether the pointed-to **mbstate_t** object describes an initial conversion state.

Return Values

mbsinit returns nonzero.

LEVEL

Level 1.

NAME

mbstring: **mbstowcs**, **wcstombs**, **mbsrtowcs**, **wcsrtombs** – multibyte string functions

SYNOPSIS

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

#include <wchar.h>

size_t mbsrtowcs(wchar_t *pwcs, const char **s, size_t n,
                 mbstate_t *ps);

size_t wcsrtombs(char *s, const wchar_t **pwcs, size_t n,
                 mbstate_t *ps);
```

DESCRIPTION

mbstowcs converts a sequence of multibyte characters from the array pointed to by *s* into a sequence of corresponding wide character codes and stores these codes into the array pointed to by *pwcs*, stopping after *n* codes are stored or a code with value zero (a converted null character) is stored.

wcstombs converts a sequence of wide character codes from the array pointed to by *pwcs* into a sequence of multibyte characters and stores these multibyte characters into the array pointed to by *s*, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. When *s* is a null pointer, then a call to **wcstombs** (*s*, *pwcs*, *n*) returns the number of bytes required to store the converted string, excluding the terminating null byte.

mbsrtowcs converts a sequence of multibyte characters that begins in the shift state described by *ps* from the array indirectly pointed to by *s* into a sequence of corresponding wide characters, which, if *pwcs* is not a null pointer, are then stored into the array pointed to by *pwcs*. Conversion continues up to and including a terminating null character, but the terminating null wide character will not be stored. Conversion stops prematurely in two cases: when a sequence of bytes is reached that does not form a valid multibyte character, or (if *pwcs* is not a null pointer) when *n* codes have been stored into the array pointed to by *pwcs*

when the next multibyte character does exceed the limit of *n* total bytes to be stored into the array pointed to by *s*. Each conversion takes place as if by a call to the **wcrtomb**.

If *s* is not a null pointer, the pointer object pointed to by *pwcs* is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted. If conversion stopped due to reaching a terminating null wide character and if *s* is not a null pointer, the resulting state described is the initial conversion state.

Return Values

If an invalid multibyte character is encountered, **mbstowcs** returns **(size_t)-1**. Otherwise, **mbstowcs** returns the number of array elements modified, not including the terminating zero code, if any. If *pwcs* is a null pointer, **mbstowcs** returns the number of elements required for the wide character code array.

If a wide character code is encountered that does not correspond to a valid multibyte character, **wcstombs** returns **(size_t)-1**. Otherwise, **wcstombs** returns the number of bytes modified, not including a terminating null character, if any. If *s* is a null pointer, **wcstombs** returns the number of bytes required for the character array.

If the input string does not begin with a valid multibyte character, an encoding error occurs for **mbsrtowcs**. In this case, it stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)-1**, but the conversion state is unchanged. Otherwise, it returns the number of multibyte characters successfully converted, which is the same as the number of array elements modified when *s* is not a null pointer.

If the first code is not a valid wide character, an encoding error occurs for **wcsrtombs**. In this case, it stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)-1**, but the conversion state is unchanged. Otherwise, it returns the number of bytes in the resulting multibyte characters sequence, which is the same as the number of array elements modified when *s* is not a null pointer.

SEE ALSO

mbchar(BA_LIB), **setlocale**(BA_OS),

LEVEL

Level 1.

NAME

memory: **memccpy**, **memchr**, **memcmp**, **memcpy**, **memmove**, **memset** – memory operations

SYNOPSIS

```
#include <string.h>

void *memccpy (void *s1, const void *s2, int c, size_t n);

void *memchr (const void *s, int c, size_t n);

int memcmp (const void *s1, const void *s2, size_t n);

void *memcpy (void *s1, const void *s2, size_t n);

void *memmove (void *s1, const void *s2, size_t n);

void *memset (void *s, int c, size_t n);
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of bytes bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

memccpy copies bytes from memory area *s2* into *s1*, stopping after the first occurrence of *c* (converted to an **unsigned char**) has been copied, or after *n* bytes have been copied, whichever comes first. It returns a pointer to the byte after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* bytes of *s2*.

memchr returns a pointer to the first occurrence of *c* (converted to an **unsigned char**) in the first *n* bytes (each interpreted as an **unsigned char**) of memory area *s*, or a null pointer if *c* does not occur.

memcmp compares its arguments, looking at the first *n* bytes (each interpreted as an **unsigned char**), and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2* when taken to be unsigned characters.

memcpy copies *n* bytes from memory area *s2* to *s1*. It returns *s1*.

memmove copies *n* bytes from memory areas *s2* to *s1*. Copying between objects that overlap will take place correctly. It returns *s1*.

memset sets the first *n* bytes in memory area *s* to the value of *c* (converted to an **unsigned char**). It returns *s*.

SEE ALSO

string (BA_LIB)

LEVEL

Level 1.

NAME

mktemp – make a unique filename

SYNOPSIS

```
char *mktemp(char *template);
```

DESCRIPTION

The function `mktemp()` replaces the contents of the string pointed to by *template* by a unique filename and returns *template*. The string in *template* should look like a filename with six trailing `X`s; `mktemp()` will replace the `X`s with a character string that can be used to create a unique filename.

RETURN VALUE

The function `mktemp()` returns the pointer *template*. If a unique name cannot be created, *template* will point to a null string.

SEE ALSO

tmpfile(BA_LIB), tmpnam(BA_LIB).

FUTURE DIRECTIONS

`NULL` will be returned if a unique name cannot be created.

LEVEL

Level 1.

NAME

mktime – converts a tm structure to a calendar time

SYNOPSIS

```
#include <sys/types.h>
#include <time.h>

time_t mktime(struct tm *timeptr);
```

DESCRIPTION

The `mktime()` function converts the time represented by the `struct tm` pointed to by *timeptr* into a calendar time (the number of seconds since 00:00:00 UTC, January 1, 1970)[see time(BA_ENV)].

In addition to computing the calendar time, `mktime()` normalizes the supplied `tm` structure. The original values of the `tm_wday` and `tm_yday` components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated in the definition of the structure. On successful completion, the values of the `tm_wday` and `tm_yday` components are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to be within the appropriate ranges. The final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

The original values of the components may be either greater than or less than the specified range. For example, a `tm_hour` of –1 means 1 hour before midnight, `tm_mday` of 0 means the day preceding the current month, and `tm_mon` of –2 means 2 months before January of `tm_year`.

If `tm_isdst` is positive, the original values are assumed to be in the alternate timezone. If it turns out that the alternate timezone is not valid for the computed calendar time, then the components are adjusted to the main timezone. Likewise, if `tm_isdst` is zero, the original values are assumed to be in the main timezone and are converted to the alternate timezone if the main timezone is not valid. If `tm_isdst` is negative, the correct timezone is determined and the components are not adjusted.

Local timezone information is used as if `mktime()` had called `tzset()`.

RETURN VALUE

The function `mktime()` returns the specified calendar time. If the calendar time cannot be represented, the function returns the value `(time_t)-1`.

SEE ALSO

ctime(BA_LIB), getenv(BA_LIB).

LEVEL

Level 1.

NAME

nl_langinfo – language information

SYNOPSIS

```
#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo(nl_item item);
```

DESCRIPTION

The `nl_langinfo()` function returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the programs locale. The manifest constant names and values of *item* are defined in `<langinfo.h>` [see langinfo(BA_ENV)].

For example:

```
nl_langinfo (ABDAY_1);
```

would return a pointer to the string `"Dim"` if the identified language was French and a French locale was correctly installed; or `"Sun"` if the identified language was English.

RETURN VALUE

If `setlocale()` [see setlocale(BA_OS)] has not been called successfully, or if `langinfo` data for a supported language is either not available or *item* is not defined therein, then `nl_langinfo` returns a pointer to the corresponding string in the C locale. In all locales, `nl_langinfo()` returns a pointer to an empty string if *item* contains an invalid setting.

USAGE

The array pointed to by the return value should not be modified by the program. Subsequent calls to `nl_langinfo()` may overwrite the array.

SEE ALSO

setlocale(BA_OS), langinfo(BA_ENV), nl_types(BA_ENV).

LEVEL

Level 1.

NAME

perror – system error messages

SYNOPSIS

```
#include <stdio.h>
void perror (const char *s);
```

DESCRIPTION

The function `perror()` produces a message on the standard error output describing the last error encountered during a call to a function.

The string pointed to by the argument *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error.

The error number is taken from the external variable `errno`, which is set when errors occur but not cleared when successful calls are made.

If given a null-string, the function `perror()` prints only the message and a new-line.

To simplify variant formatting of messages, the function `strerror()` [see `strerror(BA_LIB)`] can be used to return a pointer to the error message string associated with `errno`.

`perror()` marks for update the `st_ctime` and `st_mtime` fields of the underlying file associated with the standard error stream at some time between its successful completion and the completion of `fflush()`, `fclose()`, on `stderror()` or `exit()` or `abort()`.

`perror()` uses the UNIX System V Message Handling Facility.The *message* is retrieved from the locale-specific version of the system catalog **uxsyserr**. [See `setlocale(BA_OS)`].

USAGE

The `perror()` function is provided for ANSI compatibility.

SEE ALSO

abort(BA_OS), exit(BA_OS), fclose(BA_OS), gettxt(BA_LIB), setlocale(BA_OS), strerror(BA_LIB).

LEVEL

Level 1.

NAME

pfmt, **vpfmt** – display error message in standard format

SYNOPSIS

```
#include <pfmt.h>

int pfmt(FILE *stream, long flags, char *format, . . . /* args */);
#include <stdarg.h>
#include <pfmt.h>

int vpfmt(FILE *stream, long flags, char *format, va_list ap);
```

DESCRIPTION

pfmt

pfmt uses a format string for **printf** style formatting of *args*. The output is displayed on *stream*. **pfmt** encapsulates the output in the standard error message format.

If the **printf** format string is to be retrieved from a message database, the *format* argument must have the following structure:

[[*catalog***]:[***msgnum***]:]***defmsg*.

defmsg can only appear alone if flags include **MM_NOGET**.

catalog indicates the message database that contains the localized version of the format string. *catalog* must be limited to 14 characters. These characters must be selected from a set of all characters values, excluding **\0** (null) and the ASCII codes for **/** (slash) and **:** (colon).

msgnum must be a positive number that indicates the index of the string into the message database.

If *catalog* does not exist in the locale (specified by the last call to **setlocale** using the **LC_ALL** or **LC_MESSAGES** categories), or if the message number is out of bounds, **pfmt** attempts to retrieve the message from the **C** locale. If this second retrieval fails, **pfmt** uses the *defmsg* part of the *format* argument.

If *catalog* is omitted, **pfmt** attempts to retrieve the string from the default catalog specified by the last call to **setcat**. In this case, the *format* argument has the following structure:

*msgnum***:***defmsg*.

pfmt outputs

Message not found!! . . .

as the format string if:

catalog is not a valid catalog name as defined above

no catalog is specified (either explicitly or via **setcat**)

msgnum is not a positive number,

no message could be retrieved and *defmsg* was omitted

The *flags* determine the type of output (that is, whether the *format* should be interpreted as is or encapsulated in the standard message format), and the access to message catalogs to retrieve a localized version of *format*.

The *flags* are composed of several groups, and can take the following values (one from each group):

Output format control

MM_NOSTD do not use the standard message format, interpret *format* as a **printf** *format*. Only *catalog access control flags* should be specified if **MM_NOSTD** is used; all other flags will be ignored.

MM_STD output using the standard message format (default, value 0).

Catalog access control

MM_NOGET do not retrieve a localized version of *format*. In this case, only the *defmsg* part of the *format* is specified.

MM_GET retrieve a localized version of *format*, from the *catalog*, using *msgnum* as the index and *defmsg* as the default message (default, value 0).

Severity (standard message format only)

MM_HALT generates a localized version of **HALT**.

MM_ERROR generates a localized version of **ERROR** (default, value 0).

MM_WARNING generates a localized version of **WARNING**.

MM_INFO generates a localized version of **INFO**.

Additional severities can be defined. Add-on severities can be defined with number-string pairs with numeric values from the range [5-255], using **addsev**(BA_LIB). The numeric value ORed with other *flags* will generate the specified severity.

If the severity is not defined, **pfmt** uses the string **SEV=***N* where *N* is replaced by the integer severity value passed in *flags*.

Multiple severities passed in *flags* will not be detected as an error. Any combination of severities will be summed and the numeric value will cause the display of either a severity string (if defined) or the string **SEV**=*N* (if undefined).

Action

MM_ACTION specifies an action message. Any severity value is superseded and replaced by a localized version of **TO FIX**.

Standard Error Message Format

pfmt displays error messages in the following format:

label: *severity*: *text*

If no *label* was defined by a call to **setlabel**, the message is displayed in the format:

severity: *text*

If **pfmt** is called twice to display an error message and a helpful *action* or recovery message, the output can look like:

label: *severity*: *text*
label: TO FIX: *text*

vpfmt

vpfmt is the same as **pfmt** except that instead of being called with a variable number of arguments, it is called with an argument list as defined by the **stdarg.h** header file.

The **stdarg.h** header file defines the type **va_list** and a set of macros for advancing through a list of arguments whose number and types may vary. The argument *ap* to **vpfmt** is of type **va_list**. This argument is used with the **stdarg.h** header file macros **va_start**, **va_arg** and **va_end** [see **va_start**, **va_arg**, and **va_end** in **stdarg**(BA_ENV)]. The USAGE sections below show their use.

The macro **va_alist** is used as the parameter list in a function definition as in the function called **error** in the example below. The macro

va_start(*ap***,)**

where *ap* is of type **va_list**, must be called before any attempt to traverse and access unnamed arguments. Calls to

va_arg(*ap***,** *atype***)**

traverse the argument list. Each execution of **va_arg** expands to an expression with the value and type of the next argument in the list *ap*, which is the same object initialized by **va_start**. The argument *atype* is the type that the returned argument is expected to be.

The

va_end(*ap***)**

macro must be invoked when all desired arguments have been accessed. [The argument list in *ap* can be traversed again if **va_start** is called again after **va_end**.] In the example below, **va_arg** is executed first to retrieve the format string passed to **error**. The remaining **error** arguments, *arg1*, *arg2*, . . ., are given to **vpfmt** in the argument *ap*.

Return Values

On success, **pfmt** and **vpfmt** return the number of bytes transmitted. On failure, they return a negative value:

Errors

–1 write error to *stream*

USAGE

pfmt Example 1

```
setlabel("UX:test");
pfmt(stderr, MM_ERROR, "test:2:Cannot open file: %s\n",
  strerror(errno));
```

displays the message:

```
UX:test: ERROR: Cannot open file: No such file or directory
```

NAME

fprintf, **printf**, **snprintf**, **sprintf** – print formatted output

SYNOPSIS

```
#include <stdio.h>
int fprintf(FILE *strm, const char *format, .../* args */);
int printf(const char *format, .../* args */);
int snprintf(char *s, size_t maxsize, const char *format, .../* args */);
int sprintf(char *s, const char *format, .../* args */);
```

DESCRIPTION

Each of these functions converts, formats, and outputs its *args* under control of the character string *format*. Each function returns the number of characters transmitted (not including the terminating null character in the case of **snprintf**, and **sprintf**) or a negative value if an output error was encountered.

fprintf places output on *strm*.

printf places output on the standard output stream **stdout**.

sprintf places output, followed by a null character (**\0**), in consecutive bytes starting at *s*. It is the caller's responsibility to ensure that enough storage is available.

snprintf behaves like **sprintf**, except that no more than **maxsize** characters are placed into the array, including the terminating null character. If more than *maxsize* characters were requested, the output array will contain exactly *maxsize* characters, with a null character being the last (when *maxsize* is nonzero); a negative value is returned.

The *format* consists of zero or more ordinary characters (not **%**) which are directly copied to the output, and zero or more conversion specifications, each of which is introduced by the a **%** and results in the fetching of zero or more associated *args*.

Each conversion specification takes the following general form and sequence:

%[*pos***$][** *flags* **][** *width* **][.**_prec_ **][** *size* **]***fmt*

*pos***$** An optional entry, consisting of one or more decimal digits followed by a **$** character, that specifies the number of the next *arg* to access. The first *arg* (just after *format*) is numbered 1. If this entry is not present, the *arg* following the most recently used *arg* will be accessed.

flags Zero or more characters that change the meaning of the conversion specification. The *flag* characters and their meanings are:

- **–** The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)
- \+ The result of a signed conversion will always begin with a sign (+ or **–**). (It will begin with a sign only when a negative value is converted if this flag is not specified.)
- *space* If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

The value is to be converted to an alternate form, depending on the *fmt* character:

a, **A**, **e**, **E**, **f**, **F**, **g**, **G**
The result will contain a decimal point character, even if no digits follow. (Normally, the decimal point character is only present when fractional digits are produced.)

b, **B** A nonzero result will have **0b** or **0B** prefixed to it.

g, **G** Trailing zero digits will not be removed from the result, as they normally are.

o The precision is increased (only when necessary) to force a zero as the first digit.

x, **X** A nonzero result will have **0x** or **0X** prefixed to it.

For other conversions, the behavior is undefined.

0 For all numeric conversions (**a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**, **b**, **B**, **d**, **i**, **o**, **u**, **x** and **X**), leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **–** flags both appear, the **0** flag will be ignored. For the integer numeric conversions (**b**, **B**, **d**, **i**, **o**, **u**, **x** and **X**), if a precision is specified, the **0** flag will be ignored. For other conversions, the behavior is undefined.

´ (an apostrophe) The nonfractional portion of the result of a decimal numeric conversion (**d**, **i**, **u**, **f**, **F**, **g** and **G**) will be grouped by the current locale's thousands' separator character.

width An optional entry that consists of either one or more decimal digits, or an asterisk (*****), or an asterisk followed by one or more decimal digits and a **$**. It specifies the minimum field width: If the converted value has fewer characters than the field width, it will be padded (with space by default) on the left or right (see the above *flags* description) to the field width.

.*prec* An optional entry that consists of a period (**.**) that precedes either zero or more decimal digits, or an asterisk (*****), or an asterisk followed by one or more decimal digits and a **$**. It specifies a value that depends on the *fmt* character:

a, **A**, **e**, **E**, **f**, **F**
It specifies the number of fractional digits (those after the decimal point character). For the hexadecimal floating conversions (**a** and **A**), the number of fractional digits is just sufficient to produce an exact representation of the value (trailing zero digits are removed); for the other conversions, the default number of fractional digits is 6.

b, **B**, **d**, **i**, **o**, **u**, **x**, **X**
It specifies the minimum number of digits to appear. The default minimum number of digits is 1.

g, **G** It specifies the maximum number of significant digits. The default number of significant digits is 6.

s, **S** It specifies the maximum number of bytes to output. The default is to take all elements up to the null terminator (the entire string).

If only a period is specified, the precision is taken to be zero. For other conversions, the behavior is undefined.

size An optional **h**, **l** (ell), or **L** that specifies other than the default argument type, depending on the *fmt* character:

a, **A**, **e**, **E**, **f**, **F**, **g**, **G**
The default argument type is **double**; an **l** is ignored for compatibility with the **scanf** functions (a **float** *arg* will have been promoted to **double**); an **L** causes a **long double** *arg* to be converted.

b, **B**, **o**, **u**, **x**, **X**
The default argument type is **unsigned int**; an **h** causes the **unsigned int** *arg* to be narrowed to **unsigned short** before conversion; an **l** causes an **unsigned long** *arg* to be converted.

c The default argument type is **int** which is narrowed to **unsigned char** before output; an **l** causes a **wchar_t** *arg* to be converted (to a multibyte character). **lc** is a synonym for **C**.

d, **i** The default argument type is **int**; an **h** causes the **int** *arg* to be narrowed to **short** before conversion; an **l** causes a **long** *arg* to be converted.

n The default argument type is pointer to **int**; an **h** changes it to be a pointer to **short**, and **l** to pointer to **long**.

s The default argument type is pointer the first element of a character array; an **l** changes it to be a pointer to the first element of a **wchar_t** array. **ls** is a synonym for **S**.

If a *size* appears other than in these combinations, the behavior is undefined.

fmt A conversion character (described below) that shows the type of conversion to be applied.

When a field width or precision includes an asterisk (*****), an **int** *arg* supplies the width or precision value, and is said to be ''indirect''. A negative indirect field width value is taken as a **–** flag followed by a positive field width. A negative indirect precision value will be taken as zero. When an indirect field width or precision includes a **$**, the decimal digits similarly specify the number of the *arg* that supplies the field width or precision. Otherwise, an **int** *arg* following the most recently used *arg* will be accessed for the indirect field width, or precision, or both, in that order; the *arg* to be converted immediately follows these. Thus, if a conversion specification includes *pos***$** as well as a **$**-less indirect field width, or precision, or both, *pos* is taken to be the number of the **int** *arg* used for the first **$**-less indirection, not the *arg* to be converted.

When numbered argument specifications are used, specifying the *N*th argument requires that all the preceding arguments, from the first to the (*N*–1)th, be specified at least once, in a consistent way, in the format string.

The conversion characters and their meanings are:

a, **A** The floating *arg* is converted to hexadecimal floating notation in the style **[–]0x***h***.***hhh***p*±*d*. The binary exponent of the converted value (*d*) is one or more decimal digits. The number of fractional hexadecimal digits *h* is equal to the precision. If the precision is missing, the result will have just enough digits to represent the value exactly. The value is rounded when fewer fractional digits is specified. If the precision is zero and the **#** flag is not specified, no decimal point character appears. The single digit to the left of the decimal point character is nonzero for normal values. The **A** conversion specifier produces a value with **0X** and **P** instead of **0x** and **p**.

b, **B**, **o**, **u**, **x**, **X**
The unsigned integer *arg* is converted to unsigned binary (**b** and **B**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** and **X**). The **x** conversion uses the letters **abcdef** and the **X** conversion uses the letters **ABCDEF**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

c The integer *arg* is converted to an **unsigned char**, and the resulting character is output.

C, **lc** The wide character **wchar_t** *arg* is converted into a multibyte character and output.

d, **i** The integer *arg* is converted to signed decimal. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

e, **E** The floating *arg* is converted to the style **[–]***d***.***ddd***e**±*dd*, where there is one digit before the decimal point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal point character appears. The value is rounded to the appropriate number of digits. The **E** conversion character will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

f, **F** The floating *arg* is converted to decimal notation in the style **[–]***ddd***.***ddd*, where the number of fractional digits is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal point character appears. If a decimal point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

g, **G** The floating *arg* is converted in style **e** or **f** (or in style **E** or **F** in the case of a **G** conversion character), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted; style **e** (or **E**) will be used only if the exponent resulting from the conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point character appears only if it is followed by a digit.

n The *arg* is taken to be a pointer to an integer into which is written the number of characters output so far by this call. No argument is converted.

p The *arg* is taken to be a pointer to **void**. The value of the pointer is converted to an sequence of printable characters, which matches those read by the **%p** conversion of the **scanf**(BA_LIB) functions.

s The *arg* is taken to be a pointer to the first element of an array of characters. Characters from the array are written up to (but not including) a terminating null character; if a precision is specified, no more than that many characters are written. If a precision is not specified or is greater than the size of the array, the array must contain a terminating null character. (A null pointer for *arg* will yield undefined results.)

S, **ls** The *arg* is taken to be a pointer to the first element of an array of **wchar_t**. Wide characters from the string are converted into multibyte characters, and output until a null wide character is encountered or the number of bytes given by the precision wide would be surpassed. If the precision specification is missing, it is taken to be infinite. In no case will a partial multibyte character be output.

% Output a **%**; no argument is converted.

If the form of the conversion specification does not match any of the above, the results of the conversion are undefined. Similarly, the results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are ignored.

If a floating-point value represents an infinity, the output is [±]*inf*, where *inf* is **infinity** or **INFINITY** when the field width or precision is at least 8 and **inf** or **INF** otherwise, the uppercase versions used only for a capitol conversion character. Output of the sign follows the rules described above.

If a floating-point value has the internal representation for a NaN (not-a-number), the output is [±]*nan*[**(***m***)**]. Depending on the conversion character, *nan* is similarly either **nan** or **NAN**. If the represented NaN matches the architecture's default, no **(***m***)** will be output. Otherwise *m* represents the bits from the significand in hexadecimal with **abcdef** or **ABCDEF** used, depending on the case of the conversion character. Output of the sign follows the rules described above.

Otherwise, the locale's decimal point character will be used to introduce the fractional digits of a floating-point value.

A nonexistent or small field width does not cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result. Characters generated on streams (**stdout** or *strm*) are printed as if the **putc** function had been called repeatedly.

Errors

These functions return the number of characters transmitted (not counting the terminating null character for **sprintf**, **vsprintf**, **snprintf** and **vsnprintf**), or return a negative value if an error was encountered.

USAGE

To print a date and time in the form ''Sunday, July 3, 10:02,'' where **weekday** and **month** are pointers to null-terminated strings:

```
printf("%s, %s %i, %d:%.2d",
        weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

The following two calls to **printf** both produce the same result of **10 10 00300 10**:

```
printf("%d %1$d %.*d %1$d", 10, 5, 300);
printf("%d %1$d %3$.*2$d %1$d", 10, 5, 300);
```

The following shows a simple use of **vfprintf**, a function that writes formatted output to **stderr** by default.

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

void errprintf(FILE *fp, const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    if (fp == 0)
        fp = stderr;
    (void)vfprintf(fp, fmt, ap);
    va_end(ap);
}
```

SEE ALSO

abort(BA_OS), **exit**(BA_OS), **scanf**(BA_LIB), **fwprintf**(BA_LIB), **fwscanf**(BA_LIB), **lseek**(BA_OS), **putc**(BA_LIB), **setlocale**(BA_OS), **stdio**(BA_LIB), **write**(BA_OS)

LEVEL

Level 1.

NAME

ptsname – get name of the slave pseudo-terminal device

SYNOPSIS

```
#include <stdio.h>

char *ptsname(int fildes);
```

DESCRIPTION

The function `ptsname()` returns the name of the slave pseudo-terminal device associated with a master pseudo-terminal device. *fildes* is a file descriptor returned from a successful open of the master device. `ptsname()` returns a pointer to a string containing the null-terminated pathname of the slave device of the form `/dev/pts/`*N*.

RETURN VALUE

Upon successful completion, the function `ptsname()` returns a pointer to a string which is the name of the pseudo-terminal slave device. This value points to a static data area that is overwritten by each call to `ptsname()`. Upon failure, `ptsname()` returns `NULL`. This could occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file system.

SEE ALSO

grantpt(BA_LIB), open(BA_OS), ttyname(BA_LIB), unlockpt(BA_LIB).

LEVEL

Level 1.

NAME

putc, **putchar**, **fputc**, **putw** – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(int c, FILE *stream);

int putchar(int c);

int fputc(int c, FILE *stream);

int putw(int w, FILE *stream);
```

DESCRIPTION

putc writes *c* (converted to an **unsigned char**) onto the output *stream* at the position where the file pointer (if defined) is pointing, and advances the file pointer appropriately. If the file cannot support positioning requests, or *stream* was opened with append mode, the character is appended to the output *stream*. **putchar(c)** is defined as **putc(c, stdout)**. **putc** and **putchar** are macros.

fputc behaves like **putc**, but is a function rather than a macro. **fputc** runs more slowly than **putc**, but it takes less space per invocation and its name can be passed as an argument to a function.

putw writes the word (that is, integer) *w* to the output *stream* (where the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. **putw** neither assumes nor causes special alignment in the file.

Return Values

Upon successful completion, the functions **putc**, **fputc**, and **putchar** return the value they have written. Otherwise, these functions return the constant EOF and set **errno** to indicate the error. The function **putw** returns non-zero and sets the error indicator for the stdio-stream when an error has occurred. Otherwise, the function returns **0**.

Errors

On success, these functions (with the exception of **putw**) each return the value they have written. **putw** returns **ferror** (*stream*). Otherwise, these functions return the constant **EOF** and set **errno** to indicate the error. If a write error occurs, the error indicator for the stream is also set. This result will occur, for example, if the file *stream* is not open for writing or if the output file cannot grow. Under the following conditions, the functions **putc()**, **putchar()**, **fputc()** and **putw()** fail and set **errno** to:

EAGAIN if the **O_NONBLOCK** flag is set for the underlying file descriptor and the process would have blocked in the write operation.

EBADF if the underlying file descriptor is not a valid file descriptor open for writing.

EFBIG if an attempt was made to write a file that exceeds the process's file size limit [see ulimit(BA_OS) and getrlimit(BA_OS)].

EINTR if a signal was caught during the **putc()**, **putchar()**, **fputc()** or **putw()** call and no data was transferred.

EIO if a physical I/O error has occurred or the process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU** and the process group of the process is orphaned.

ENOSPC if there is no free space remaining on the device containing the file.

ENXIO if the device associated with the underlying file descriptor is a block-special or character-special file and the file-pointer value is out of range.

EPIPE if an attempt is made to write to a FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

SEE ALSO

abort(BA_OS), **fclose**(BA_OS), **ferror**(BA_OS), **fopen**(BA_OS), **fread**(BA_OS), **ftrylockfile**(MT_LIB), **flockfile**(MT_LIB), **printf**(BA_LIB), **puts**(BA_LIB), **setbuf**(BA_LIB), **stdio**(BA_LIB),

LEVEL

Level 1.

NOTICES

Because it is implemented as a macro, **putc** evaluates a *stream* argument more than once. In particular, **putc(c, *f++);** doesn't work sensibly. **fputc** should be used instead.

Because of possible differences in word length and byte ordering, files written using **putw** are machine-dependent, and may not be read using **getw** on a different processor.

Functions exist for all the above defined macros. To get the function form, the macro name must be undefined (for example, **#undef putc**).

NAME

putenv – change or add value to environment

SYNOPSIS

```
#include <stdlib.h>

int putenv(char *string);
```

DESCRIPTION

The argument *string* points to a string of the the following form:

name=value

The function `putenv()` makes the `value` of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to the function `putenv()`.

RETURN VALUE

The function `putenv()` returns non-zero if it was unable to obtain enough space for an expanded environment, otherwise zero.

USAGE

The function `putenv()` manipulates the environment pointed to by `environ`, and can be used in conjunction with `getenv()`. However, *envp*, the third argument to `main()`, is not changed [see exec(BA_OS)].

A potential error is to call the function `putenv()` with a pointer to an automatic variable as the argument and to then exit the calling function while *string* is still part of the environment.

SEE ALSO

exec(BA_OS), malloc(BA_OS), getenv(BA_LIB).

LEVEL

Level 1.

NAME

puts, fputs – put a string on a stdio-stream

SYNOPSIS

```
#include <stdio.h>

int puts(const char *s);

int fputs(const char *s, FILE *strm);
```

DESCRIPTION

The function `puts()` writes the null-terminated string pointed to by *s*, followed by a newline character, to the standard output stream `stdout`.

The function `fputs()` writes the null-terminated string pointed to by *s* to *strm*.

Neither function writes the terminating null character.

The `st_ctime` and `st_mtime` fields of the file will be marked for update between the successful execution of `puts()` or `fputs()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

RETURN VALUE

Upon successful completion, the functions `puts()` and `fputs()` return the number of characters written; otherwise these functions return `EOF` and set `errno` to indicate an error.

ERRORS

Under the following conditions, the functions `puts()`, and `fputs()` fail and set `errno` to:

`EAGAIN` if the `O_NONBLOCK` flag is set for the underlying file descriptor and the process would have blocked in the write operation.

`EBADF` if the underlying file descriptor is not a valid file descriptor open for writing.

`EFBIG` if an attempt was made to write a file that exceeds the process's file size limit [see ulimit(BA_OS) and getrlimit(BA_OS)].

`EINTR` if a signal was caught during the `puts()`, or `fputs()` call and no data was transferred.

`EIO` if a physical I/O error has occurred or the process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU` and the process group of the process is orphaned.

`ENOSPC` if there is no free space remaining on the device containing the file.

`ENXIO` if the device associated with the underlying file descriptor is a block-special or character-special file and the file-pointer value is out of range.

`EPIPE` if an attempt is made to write to a FIFO that is not open for reading by any process. A `SIGPIPE` signal is also sent to the process.

USAGE

The function `puts()` appends a newline character while `fputs()` does not.

SEE ALSO

ferror(BA_OS), fopen(BA_OS), fread(BA_OS), gets(BA_LIB), printf(BA_LIB), putc(BA_LIB).

LEVEL

Level 1.

NAME

putwc, **putwchar**, **fputwc** – put wide character on a stream

SYNOPSIS

```
#include <stdio.h>
#include <widec.h>

wint_t putwc(wint_t c, FILE *stream);

wint_t putwchar(wint_t c);

wint_t fputwc(wint_t c, FILE *stream);
```

DESCRIPTION

putwc transforms the wide character *c* into a multibyte character, and writes it to the output stream (at the position where the file pointer, if defined, is pointing). **putwchar(c)** is equivalent to **putwc(c, stdout)**.

putwc behaves like **fputwc**, expect that **putwc** may be implemented as a macro that evaluates *stream* more than once.

Errors

On success, these functions return the value they have written. On failure, they return the constant **WEOF**. If an I/O error occurs, the error indicator is set for the stream. If **c** does not correspond to a valid multibyte character, **errno** will be set to **EILSEQ**.

SEE ALSO

fclose(BA_OS), **ferror**(BA_OS), **fopen**(BA_OS), **printf**(BA_LIB), **setbuf**(BA_LIB), **stdio**(BA_LIB)

LEVEL

Level 1.

NAME

fputws – put a **wchar_t** string on a stream

SYNOPSIS

```
#include <stdio.h>
#include <widec.h>

int fputws(const wchar_t *s, FILE *stream);
```

DESCRIPTION

fputws transforms the **wchar_t** null-terminated **wchar_t** string pointed to by *s* into a multibyte character string, and writes the string to the named output stream. This function does not write the terminating **wchar_t** null character.

Errors

On success, this function returns the number of **wchar_t** characters transformed and written. Otherwise it returns **EOF**.

SEE ALSO

fread(BA_OS), **printf**(BA_LIB), **putwc**(BA_LIB), **stdio**(BA_LIB)

LEVEL

Level 1.

NAME

qsort – quicker sort

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
        int (*compar)(const void *, const void *));
```

DESCRIPTION

The function `qsort()` is a general sorting algorithm. It sorts a table of data in place. The contents of the table are sorted in ascending order according to the user supplied comparison function.

The argument *base* points to the element at the base of the table.

The argument *nel* is the number of elements in the table.

The argument *width* is the size of an element in bytes.

The argument *compar* is the name of the user supplied comparison function, which is called with two arguments that point to the elements being compared. The comparison function must return an integer less than, equal to or greater than zero to indicate if the first argument is to be considered less than, equal to or greater than the second argument.

USAGE

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The relative order in the output of two items which compare as equal is unpredictable.

SEE ALSO

bsearch(BA_LIB), lsearch(BA_LIB), string(BA_LIB).

LEVEL

Level 1.

NAME

rand, **srand** – simple random-number generator

SYNOPSIS

```
#include <stdlib.h>

int rand(void);

void srand(unsigned int seed);
```

DESCRIPTION

rand uses a multiplicative congruent random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to **RAND_MAX** (defined in **stdlib.h**).

The function **srand** uses the argument *seed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to the function **rand**. If the function **srand** is then called with the same *seed* value, the sequence of pseudo-random numbers will be repeated. If the function **rand** is called before any calls to **srand** have been made, the same sequence will be generated as when **srand** is first called with a *seed* value of 1.

SEE ALSO

drand48(BA_LIB)

LEVEL

Level 2: September 30, 1989.
*Level 2: June 1993.

NOTICES

The spectral properties of **rand** are limited. **drand48**(BA_LIB) provides a much better, though more elaborate, random-number generator.

Each thread that accesses one of the functions **drand48, lrand48, mrand48, srand48, seed48**, or **lcong48** should be coded as per the following example:

```
mutex_lock(I_am_using_drand48);
value = FUNCTION();
mutex_unlock(I_am_using_drand48);
```

where **FUNCTION** is one of those listed. The same mutex must be used for all six functions.

NAME

regcomp,regexec,regerror,regfree – regular expression matching

SYNOPSIS

```
#include <regex.h>

int regcomp(regex_t *preg,const char *pattern, int flags);

int regexec(const regex_t *preg,const char *string, size_t n,
        regmatch_t *pmatch, int flags);

size_t regerror(int ecode, const regex_t *preg, char *buf, size_t n);

void regfree(regex_t *preg);
```

DESCRIPTION

These functions are part of the X/Open Portability Guide Issue 4 optional POSIX2 C-Language Binding feature group.

Return Values

regcomp returns **REG_NOSYS** and sets **errno** to **ENOSYS**.

regerror returns **0** and sets **errno** to **ENOSYS**.

regexec returns **REG_NOSYS** and sets **errno** to **ENOSYS**.

regfree returns and sets **errno** to **ENOSYS**.

USAGE

Administrator.

SEE ALSO

regexp(BA_ENV)

LEVEL

Level 1.

NAME

regexp: compile, step, advance – regular expression compile and match routines

SYNOPSIS

```
#define INIT declarations
#define GETC(void) getc code
#define PEEKC(void) peekc code
#define UNGETC(void) ungetc code
#define RETURN(ptr) return code
#define ERROR(val) error code

#include <regexp.h>

char *compile(char *instring, char *expbuf, char *endbuf,
     int eof);

int step(char *string, char *expbuf);

int advance(char *string, char *expbuf);

extern char *loc1, *loc2, *locs;
```

DESCRIPTION

These functions are general purpose regular expression matching routines to be used in programs that perform regular expression matching. These functions are defined by the `<regexp.h>` header file.

The functions `step()` and `advance()` do pattern matching given a character string and a compiled regular expression as input.

The function `compile()` takes as input a regular expression as defined below and produces a compiled expression that can be used with `step()` or `advance()`.

A regular expression specifies a set of character strings. A member of this set of strings is said to be matched by the regular expression. Some characters have special meaning when used in a regular expression; other characters stand for themselves.

The regular expressions available for use with the regexp functions are constructed as follows:

Expression	*Meaning*
c	the character *c* where *c* is not a special character.
c	the character *c* where *c* is any character, except a digit in the range `1-9`.
^	the beginning of the line being compared.
$	the end of the line being compared.
.	any character in the input.
[*s*]	any character in the set *s*, where *s* is a sequence of characters and/or a range of characters, *e.g.*, [*c*-*c*].

`[^s]` any character not in the set *s*, where *s* is defined as above.

*r** zero or more successive occurrences of the regular expression *r*. The longest leftmost match is chosen.

rx the occurrence of regular expression *r* followed by the occurrence of regular expression *x*. (Concatenation)

r`\{`*m*`,`*n*`\}` any number of *m* through *n* successive occurrences of the regular expression *r*. The regular expression *r*`\{`*m*`\}` matches exactly *m* occurrences; *r*`\{`*m*`,\}` matches at least *m* occurrences.

`\(`*r*`\)` the regular expression *r*. When `\`*n* (where *n* is a number greater than zero) appears in a constructed regular expression, it stands for the regular expression *x* where *x* is the n^{th} regular expression enclosed in `\(` and `\)` that appeared earlier in the constructed regular expression. For example, `\(`*r*`\)`*x*`\(`*y*`\)`*z*`\2` is the concatenation of regular expressions *rxyzy*.

Characters that have special meaning except when they appear within square brackets (`[ ]`) or are preceded by `\` are: `.`, `*`, `[`, `\`. Other special characters, such as `$` have special meaning in more restricted contexts.

The character `^` at the beginning of an expression permits a successful match only immediately after a newline, and the character `$` at the end of an expression requires a trailing newline.

Two characters have special meaning only when used within square brackets. The character `-` denotes a range, `[`*c*`-`*c*`]`, unless it is just after the open bracket or before the closing bracket, `[-`*c*`]` or `[`*c*`-]` in which case it has no special meaning. When used within brackets, the character `^` has the meaning *complement of* if it immediately follows the open bracket (example: `[^`*c*`]`); elsewhere between brackets (example: `[`*c*`^]`) it stands for the ordinary character `^`.

The special meaning of the `\` operator can be escaped only by preceding it with another `\`, *e.g.* `\\`.

Programs must have the following five macros declared before the `#include <regexp.h>` statement. These macros are used by the `compile()` routine. The macros `GETC()`, `PEEKC()`, and `UNGETC()` operate on the regular expression given as input to `compile()`.

`GETC()` This macro returns the value of the next character (byte) in the regular expression pattern. Successive calls to `GETC()` should return successive characters of the regular expression.

`PEEKC()` This macro returns the next character (byte) in the regular expression. Immediately successive calls to `PEEKC()` should return the same character, which should also be the next character returned by `GETC()`.

`UNGETC()` This macro causes the argument `c` to be returned by the next call to `GETC()` and `PEEKC()`. No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by `GETC()`. The return value of the macro `UNGETC(c)` is always ignored.

`RETURN(`*ptr*`)` This macro is used on normal exit of the `compile()` routine. The value of the argument *ptr* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.

`ERROR(`*val*`)` This macro is the abnormal return from the `compile()` routine. The argument *val* is an error number [see **ERRORS** below for meanings]. This call should never return.

The syntax of the `compile()` routine is as follows:

`compile(`*instring*`, `*expbuf*`, `*endbuf*`, `*eof*`)`

The first parameter, *instring*, is never used explicitly by the `compile()` routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the `INIT` declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of `(char *)0` for this parameter.

The next parameter, *expbuf*, is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in `(endbuf-expbuf)` bytes, a call to `ERROR(50)` is made.

The parameter *eof* is the character which marks the end of the regular expression. This character is usually a `/`.

Each program that includes the `<regexp.h>` header file must have a `#define` statement for `INIT`. It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for `GETC()`, `PEEKC()`, and `UNGETC()`. Otherwise it can be used to declare external variables that might be used by `GETC()`, `PEEKC()` and `UNGETC()`. [See **EXAMPLE** below.]

The first parameter to the `step()` and `advance()` functions is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter, *expbuf*, is the compiled regular expression which was obtained by a call to the function `compile()`.

The function `step()` returns non-zero if some substring of *string* matches the regular expression in *expbuf* and zero if there is no match. If there is a match, two external character pointers are set as a side effect to the call to `step()`. The variable `loc1` points to the first character that matched the regular expression; the variable `loc2` points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire input string, `loc1` will point to the first character of *string* and `loc2` will point to the null at the end of *string*.

The function `advance()` returns non-zero if the initial substring of *string* matches the regular expression in *expbuf*. If there is a match, an external character pointer, `loc2`, is set as a side effect. The variable `loc2` points to the next character in *string* after the last character that matched.

When `advance()` encounters a * or `\{ \}` sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, `advance()` will back up along the string until it finds a match or reaches the point in the string that initially matched the * or `\{ \}`. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer `locs` is equal to the point in the string at sometime during the backing up process, `advance()` will break out of the loop that backs up and will return zero.

The external variables `circf`, `sed`, and `nbra` are reserved.

RETURN VALUE

The function `compile()` uses the macro `RETURN` on success and the macro `ERROR` on failure (see above). The functions `step()` and `advance()` return non-zero on a successful match and zero if there is no match.

ERRORS

`11` range endpoint too large.

`16` bad number.

`25` \ *digit* out of range.

`36` illegal or missing delimiter.

`41` no remembered search string.

`42` `\( \)` imbalance.

`43` too many `\(`.

`44` more than 2 numbers given in `\{ \}`.

`45` `}` expected after \.

`46` first number exceeds second in `\{ \}`.

`49` `[ ]` imbalance.

`50` regular expression overflow.

EXAMPLE

The following is an example of how the regular expression macros and calls might be defined by an application program:

```
#define INIT         register char *sp = instring;
#define GETC()       (*sp++)
#define PEEKC()      (*sp)
#define UNGETC(c)    (--sp)
#define RETURN(*c)    return;
#define ERROR(c)     regerr()

#include <regexp.h>

 . . .
     (void) compile(*argv, expbuf, &expbuf[ESIZE],'\0');
 . . .
     if (step(linebuf, expbuf))
                       succeed();
```

FUTURE DIRECTIONS

The functionality of the regexp functions will eventually be replaced by a more complete interface and the regexp functions will be discontinued.

LEVEL

Level 2: September 30, 1989.

NAME

scalb, logb, nextafter – radix-independent functions

SYNOPSIS

```
#include <math.h>

‡double scalb(double x, double n);

‡double logb(double x);

double nextafter(double x, double y);
```

DESCRIPTION

The functions `scalb()`, `logb()`, and `nextafter()` supply radix-independent facilities for manipulating floating point numbers.

The function `scalb()` returns $x * r^n$ where r is the radix of the machine's floating point arithmetic. When r is 2, `scalb()` returns the same value as `ldexp` [see `ldexp()` in frexp(BA_LIB)].

The function `logb()` returns the exponent of x. Formally, the return value is the integral part of $\log_r |x|$ as a signed floating point value, for non-zero x.

The function `nextafter()` returns the next representable double-precision floating-point value following x in the direction of y. Thus, if y is less than x, `nextafter` returns the largest representable floating-point number less than x.

RETURN VALUE

A macro `HUGE_VAL` is defined in the `<math.h>` header file. This macro calls a function that either returns $+\infty$ on a system supporting the IEEE 754 standard or +{MAXDOUBLE} on a system that does not support the IEEE 754 standard.

If the correct value would overflow, the function `scalb()` returns ±`HUGE_VAL` (according to the sign of x) and sets `errno` to `ERANGE`.

If the correct value would underflow, the function `scalb()` returns zero and sets `errno` to `ERANGE`.

The function `logb()` returns –`HUGE_VAL` when x is zero and sets `errno` to `EDOM`.

On implementations which support IEEE `NaN`, if an input parameter is `NaN`, then the function will return `NaN`.

SEE ALSO

frexp(BA_LIB).

FUTURE DIRECTIONS

In a future edition of the SVID, logb will be updated according to NCEG recommendations to be conformant to the IEEE Standard 854 rather than 754.

LEVEL

Level 1.

`logb()` is designated Level 2, June 1993.

`scalb()` is designated Level 2, September 30, 1993.

NAME

fscanf, **scanf**, **sscanf** – convert formatted input

SYNOPSIS

```
#include <stdio.h>
int fscanf(FILE *strm, const char *format, .../* args */);
int scanf(const char *format, .../* args */);
int sscanf(const char *s, const char *format, .../* args */);
```

DESCRIPTION

Each function reads characters, interprets them, and stores the results through the *arg* pointers, under control of the character string *format*. Each function returns the number of successfully matched and assigned input items, or **EOF** if the input ended prior any successful matches.

fscanf reads from the stream *strm*.

scanf reads from the standard input stream, **stdin**.

sscanf reads from the character string *s*.

The *format* consists of zero or more portable white-space characters (blanks, horizontal and vertical tabs, newlines, carriage returns, and form-feeds) which cause single-byte white-space input characters [as defined by **isspace**, see **ctype**(BA_LIB)] to be skipped, zero or more ordinary characters (not **%**) which must match the next input characters, and zero or more conversion specifications, each of which is introduced by the a **%** which can result in the matching of a sequence of input characters and possibly the assignment of a converted value.

Each conversion specification takes the following general form and sequence:

%[*pos***$][*******][** *width***][** *size*

b, **o**, **u**, **x**
The default argument type is pointer to **unsigned int**; an **h** changes it to be a pointer to **unsigned short int**, and **l** to pointer to **unsigned long int**.

c, **s**, **[...]**
The default argument type is pointer to character; an **l** changes it to a pointer to **wchar_t**. **lc** (**ls**) is a synonym for **C** (**S**).

d, **i**, **n** The default argument type is pointer to **int**; an **h** changes it to be a pointer to **short int**, and **l** to pointer to **long int**.

If a *size* appears other than in these combinations, the behavior is undefined.

fmt A conversion character or sequence (described below) that shows the type of conversion to be applied.

A conversion specification directs the matching and conversion of the next input item; the result is placed in the object pointed to by the corresponding *arg* unless assignment suppression was indicated by the * flag. The suppression of assignment provides a way of describing an input item that is to be skipped. For all conversion specifiers except **c**, **C**, **n** and **[...]**, leading single-byte white-space characters are skipped. An input item is usually defined as a sequence of non-white-space single-byte characters that extends to the next inappropriate single-byte character or until the maximum field width (if one is specified) is exhausted. For **C**, **S** and **l[...]**, the field width instead specifies the number of multibyte characters.

The conversion specifiers and their meanings are:

a, **e**, **f**, **g**
Matches an optionally signed floating number, whose format is the same as expected for the subject string of the **strtod** function see **strtol**(BA_LIB).

b, **o**, **u**, **x**
Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtoul** function (see **strtol**(BA_LIB)) with the respective values of 2, 8, 10 or 16 for the *base* argument.

c Matches a sequence of single-byte characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument should be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added. The normal skip over white space is suppressed.

C, **lc** Matches a sequence of multibyte characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument should be a pointer to the initial element of a **wchar_t** array large enough to accept the sequence of generated wide characters. No null wide character is added. The normal skip over white space is suppressed.

d, **i** Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol**(BA_LIB) function with the respective values of 10 or 0 for the *base* argument.

n No input is consumed. The number of characters so far read by this call is written into the integer pointed to by the corresponding argument. Execution of a **%n** directive does not increment the assignment count returned at the completion of this call.

p Matches a sequence of printable characters as is produced by the **printf**(BA_LIB) functions' **%p** conversion. The corresponding argument should be a pointer to a pointer to **void**. If the input matched is a value converted earlier (during the same program execution), the pointer that results will compare equal to that value; otherwise, the behavior is undefined.

s Matches a sequence of single-byte characters, optionally delimited by single-byte white-space characters. The corresponding argument should be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

S, **ls** Matches a sequence of multibyte characters, optionally delimited by single-byte white-space characters. The corresponding argument should be a pointer to the initial element of a **wchar_t** array large enough to accept the sequence of generated wide characters and a terminating null wide character, which will be added automatically.

[...] Matches a nonempty sequence of single-byte characters from a set of expected characters (the *scanset*) as designated by the characters between the brackets (the *scanlist*), see below. The corresponding argument should be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

l[...]
Matches a nonempty sequence of multibyte characters from a set of expected multibyte characters (the *scanset*) as designated by the multibyte characters between the brackets (the *scanlist*), see below. The corresponding argument should be a pointer to the initial element of a **wchar_t** array large enough to accept the sequence of generated wide characters and a terminating null wide character, which will be added automatically.

% Matches a single **%**; no assignment is done.

For **[...]** and **l[...]**, the scanlist consists of all characters up to, but not including, the matching right bracket (**]**). The first right bracket matches unless the specifier begins with **[]** or **[^]**, in which case the scanlist includes a **]** and the matching one is the second right bracket. The scanset is those characters described by the scanlist unless it begins with a circumflex (^), in which case the scanset is those characters not described by the scanlist that follows the circumflex. The scanlist can describe an inclusive range of characters by *low***-***high* where *low* is not lexically greater than *high* (and where these endpoints are in the same codeset for **l[...]** in locales whose multibyte characters have such); otherwise, a dash (**-**) will stand for itself, as it will when it occurs last in the scanlist, or the first, or the second when a circumflex is first.

If the form of the conversion specification does not match any of the above, the results of the conversion are undefined. Similarly, the results are undefined if there are insufficient pointer *args* for the format. If the format is exhausted while *args* remain, the excess *args* are ignored.

When matching floating numbers, the locale's decimal point character is taken to introduce a fractional portion, the sequences **inf** and **infinity** (case ignored) are taken to represent infinities, and the sequence **nan**[**(***m***)**] (case ignored), where the optional parenthesized *m* consists of zero or more alphanumeric or underscore (_) characters, are taken to represent NaNs (not-a-numbers). Note, however, that the locale's thousands' separator character will not be recognized as such.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including newline characters) is left unread unless matched by a directive.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than **%n**, if any) is terminated with an input failure.

If a truncated sequence (due to reaching end-of-file or a conflicting input character, or because a field width is exhausted) does not form a valid match for the current directive, the directive is terminated with a matching failure.

The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

Characters from streams (**stdin** or *strm*) are read as if the **getc** function had been called repeatedly.

Errors

These routines return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, **EOF** is returned.

USAGE

The call to the function **scanf**:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to **n** the value **3**, to **i** the value **25**, to **x** the value **5.432**, and **name** will contain **thompson\0**.

The call to the function **scanf**:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0-9]", &i, &x, name);
```

with the input line:

```
56789 0123 56a72
```

will assign **56** to **i**, **789.0** to **x**, skip **0123**, and place the characters **56\0** in **name**. The next character read from **stdin** will be **a**.

The following shows a simple use of **vfscanf**, a function that reads formatted input from its own connection to **/dev/tty**.

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

static FILE *instream;

int scan(const char *fmt, ...)
{
    va_list ap;
    int ret;

    va_start(ap, fmt);
    if (instream == 0) {
        if ((instream = fopen("/dev/tty", "r")) == 0)
            return EOF;
    }
    ret = vfscanf(instream, fmt, ap);
    va_end(ap);
    return ret;
}
```

SEE ALSO

printf(BA_LIB), **fwprintf**(BA_LIB), **fwscanf**(BA_LIB), **getc**(BA_LIB), **stdio**(BA_LIB), **strtol**(BA_LIB)

LEVEL

Level 1.

NAME

setbuf, setvbuf – assign buffering to a stdio-stream

SYNOPSIS

```
#include <stdio.h>

void setbuf(FILE *strm, char *buf);

int setvbuf(FILE *strm, char *buf, int type, size_t size);
```

DESCRIPTION

The function `setbuf()` may be used after a stdio-stream has been opened, but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is `NULL`, input/output will be completely unbuffered.

A constant `BUFSIZ`, defined by the `<stdio.h>` header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

The function `setvbuf()` may be used after *strm* has been opened, but before it is read or written. The value of *type* determines how *strm* will be buffered. Legal values for *type*, defined by the `<stdio.h>` header file, are:

`_IOFBF` causes input/output to be fully buffered.

`_IOLBF` causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.

`_IONBF` causes input/output to be completely unbuffered.

If *buf* is not `NULL`, the array it points to will be used for buffering instead of an automatically allocated buffer. The value of *size* specifies the size of the buffer to be used. The constant `BUFSIZ`, in the `<stdio.h>` header file, is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

When *strm* is unbuffered, characters are intended to appear from the source or at the destination as soon as possible. Otherwise, characters may be accumulated and transmitted to and from the host environment as a block. When *strm* is fully buffered, characters are intended to be transmitted to or from the host environment as a block when the buffer is filled. When *strm* is line buffered, characters are intended to be transmitted to or from the host environment as a block when a newline character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on a line-buffered *strm* that requires the transmission of characters from the host environment.

By default, output to a terminal is line buffered and all other input/output is fully buffered, except the standard error stream `stderr`, which is normally not buffered.

RETURN VALUE

If an illegal value for *type* or *size* is provided, the function `setvbuf()` returns a non-zero value; otherwise, the value returned will be zero.

USAGE

A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stdio-stream in the same block.

SEE ALSO

fopen(BA_OS), malloc(BA_OS), getc(BA_LIB), putc(BA_LIB).

LEVEL

Level 1.

NAME

setcat – define default catalog

SYNOPSIS

```
#include <pfmt.h>

char *setcat(const char *catalog);
```

DESCRIPTION

The routine `setcat()` defines the default message catalog to be used by subsequent calls to `pfmt()`, `vpfmt()`, `lfmt()`,`vlfmt()`, or `gettxt()` that do not explicitly specify a message catalog.

catalog must be limited to 14 characters. These characters must be selected from a set of all characters values, excluding `\0` (null) and the ASCII codes for `/` (slash) and `:` (colon).

`setcat()` assumes that the catalog exists. No checking is done on the argument.

A NULL pointer passed as an argument will result in the return of a pointer to the current default message catalog name. A pointer to an empty string passed as an argument will cancel the default catalog.

If no default catalog is specified, or if *catalog* is an invalid catalog name, subsequent calls to `gettxt()`, `pfmt()`, `vpfmt()`, `lfmt()`, or `vlfmt()` that do not explicitly specify a catalog name will use `Message not found!!\n` as the default string.

RETURN VALUE

Upon success, `setcat()` returns a pointer to the catalog name. Upon failure, `setcat()` returns a NULL pointer.

EXAMPLE

```
setcat("test");
gettxt(":10", "hello world\n");
```

SEE ALSO

envvar(BA_ENV), gettxt(BA_LIB), lfmt(BA_LIB), pfmt(BA_LIB), setlocale(BA_LIB).

LEVEL

Level 2: April 1991.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp(jmp_buf env);

void longjmp(jmp_buf env, int val);
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

The function `setjmp()` saves its stack environment in *env* (whose type, `jmp_buf`, is defined by the `<setjmp.h>` header file) for later use by the function `longjmp()`. The function `setjmp()` returns the value `0`.

The function `longjmp()` restores the environment saved by the last call to the function `setjmp()` with the corresponding argument *env*.

After the function `longjmp()` is completed, program execution continues as if the corresponding call to the function `setjmp()` (the caller of which must not itself have returned in the interim) had just returned the value *val*. All accessible variables of storage class static or external have values as of the time the function `longjmp()` was called. The values of variables of storage class automatic or register are indeterminate.

RETURN VALUE

When the function `setjmp()` has been called by the calling process, it returns `0`.

The function `longjmp()` does not return from where it was called, but rather, program execution continues as if the previous call to the function `setjmp()` returned with a return value of *val*. That is, when the function `setjmp()` returns as a result of the function `longjmp()` being called, the function `setjmp()` returns *val*. However, the function `longjmp()` cannot cause the function `setjmp()` to return the value `0`. If the function `longjmp()` is invoked with a *val* of `0`, the function `setjmp()` will return `1`.

USAGE

If the function `longjmp()` is called even though the argument *env* was never primed by a call to the function `setjmp()`, or when the last such call was in a function which has since returned, the behavior is undefined.

SEE ALSO

signal(BA_OS), sigsetjmp(BA_OS).

LEVEL

Level 1.

NAME

setlabel – define the label for `pfmt()` and `lfmt()`.

SYNOPSIS

```
#include <pfmt.h>

int setlabel(const char *label);
```

DESCRIPTION

The routine `setlabel()` defines the label for messages produced in standard format by subsequent calls to `pfmt()`,`vpfmt()`, `lfmt()`, and `vlfmt()`.

label is a character string no more than 25 characters in length.

No label is defined before `setlabel()` is called. A NULL pointer or an empty string passed as argument will reset the definition of the label to no label.

RETURN VALUE

`setlabel()` returns 0 in case of success, non-zero otherwise.

USAGE

The label should be set once at the beginning of a utility and remain constant.

If `setlabel()` is called before `getopt()`, `getopt()` will use that label. Otherwise, `getopt()` will use the name of the utility.

EXAMPLE

The following code (without previous call to `setlabel()`):

```
pfmt(stderr, MM_ERROR, "test:2:Cannot open file\n");
setlabel("UX:test");
pfmt(stderr, MM_ERROR, "test:2:Cannot open file\n");
```

will produce the following output:

```
ERROR: Cannot open file
UX:test: ERROR: Cannot open file
```

SEE ALSO

getopt(BA_LIB), lfmt(BA_LIB), pfmt(BA_LIB).

LEVEL

Level 2: April 1991.

NAME

sigsetjmp, siglongjmp – a non-local goto with signal state

SYNOPSIS

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);

void siglongjmp(sigjmp_buf env, int val);
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

The function `sigsetjmp()` saves the calling process's registers, stack environment [see sigaltstack(BA_OS)] and, if *savemask* is non-zero, signal mask [see sigprocmask(BA_OS)] in *env* (whose type, `sigjmp_buf`, is defined in the `<setjmp.h>` header file) for later use by `siglongjmp()`.

The function `siglongjmp()` restores the environment saved by the last call of `sigsetjmp()` with the corresponding *env* argument. After `siglongjmp()` is completed, program execution continues as if the corresponding call of `sigsetjmp()` (which must not itself have returned in the interim) had just returned the value *val*. `siglongjmp()` cannot cause `sigsetjmp()` to return the value `0`. If `siglongjmp()` is invoked with a second argument of `0`, `sigsetjmp()` will return `1`. At the time of the second return from `sigsetjmp()`, all external and static variables have values as of the time `siglongjmp()` was called. The values of register and automatic variables are undefined.

If a signal-catching function interrupts `sleep()` and calls `siglongjmp()` to restore an environment saved prior to the `sleep()` call, the action associated with `SIGALRM` and time it is scheduled to be generated are unspecified. It is also unspecified whether the `SIGALRM` signal is blocked, unless the process's signal mask is restored as part of the environment.

The function `siglongjmp()` restores the saved signal mask if and only if the *env* argument was initialized by a call to the `sigsetjmp()` function with a non-zero *savemask* argument.

RETURN VALUE

The function `sigsetjmp()` returns the value `0` when *env* is originally established, and *val* when *env* is restored by a subsequent call to `siglongjmp()`.

The function `siglongjmp()` does not return.

SEE ALSO

sigaction(BA_OS), sigaltstack(BA_OS), sigprocmask(BA_OS), setjmp(BA_LIB). sleep(BA_OS).

LEVEL

Level 1.

NAME

stdio – standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>
FILE *stdin, *stdout, *stderr;
```

DESCRIPTION

The functions described as Standard I/O routines (`stdio`) constitute an efficient, user-level I/O buffering scheme. The functions `getc()` and `putc()` handle characters quickly. The functions `getchar()` and `putchar()`, and the higher-level routines `fgetc()`, `fgets()`, `fprintf()`, `fputc()`, `fputs()`, `fread()`, `fscanf()`, `fwrite()`, `gets()`, `getw()`, `printf()`, `puts()`, `putw()`, and `scanf()` all use or act as if they use `getc()` and `putc()`; they can be freely intermixed.

A file with associated buffering is called a stdio-stream and is declared to be a pointer to a defined type `FILE`. `fopen()` creates certain descriptive data for a stdio-stream and returns a pointer to designate the stdio-stream in all further transactions. Normally, there are three open stdio-streams with constant pointers declared in the `<stdio.h>` header file and associated with the standard open files:

`stdin`	standard input file
`stdout`	standard output file
`stderr`	standard error file

When opened, the standard error stdio-stream is not fully buffered [see setbuf(BA_LIB)]; the standard input and standard output stdio-streams are fully buffered if and only if the stdio-stream can be determined not to refer to an interactive device.

The following symbolic values in `<unistd.h>` define the file descriptors that will be associated with the C-language `stdin`, `stdout` and `stderr` when the application is started:

`STDIN_FILENO`	Standard input value, `stdin`. It has a value of `0`.
`STDOUT_FILENO`	Standard output value, `stdout`. It has a value of `1`.
`STDERR_FILENO`	Standard error value, `stderr`. It has a value of `2`.

A constant `NULL` designates a nonexistent pointer.

An integer constant `EOF` is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant `BUFSIZ` specifies the size of the buffers used by the particular implementation.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

The Standard I/O related functions and constants are declared in that header file and need no further declaration. The constants and the following ''functions'' may be implemented as macros, hence, redeclaration of these names is

perilous: `getc()`, `getchar()`, `putc()`, `putchar()`, `ferror()`, `feof()`, `clearerr()`, and `fileno()`.

RETURN VALUE

Invalid stdio-stream pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

SEE ALSO

fclose(BA_OS), ferror(BA_OS), fopen(BA_OS), fread(BA_OS), fseek(BA_OS), getc(BA_LIB), gets(BA_LIB), popen(BA_LIB), printf(BA_LIB), putc(BA_LIB), puts(BA_LIB), scanf(BA_LIB), setbuf(BA_LIB), tmpfile(BA_LIB), ungetc(BA_LIB), unistd.h(BA_ENV).

LEVEL

Level 1.

NAME

strcoll – string collation

SYNOPSIS

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

DESCRIPTION

The function `strcoll()` returns an integer greater than, equal to, or less than zero in direct correlation to whether string *s1* is greater than, equal to, or less than the string *s2*. The comparison is based on strings interpreted as appropriate to the program's locale for category `LC_COLLATE` [see setlocale(BA_OS)].

Both `strcoll()` and `strxfrm()` provide for locale-specific string sorting. `strcoll()` is intended for applications in which the number of comparisons per string is small. When strings are to be compared a number of times, `strxfrm()` is a more appropriate utility because the transformation process occurs only once.

RETURN VALUE

Upon successful completion, the `strcoll()` function returns an integer greater than, equal to or less than zero to indicate whether the string pointed to by *s1* is greater than, equal to or less than the string pointed to by *s2*, when both are interpreted as appropriate for the current locale.

SEE ALSO

setlocale(BA_OS), string(BA_LIB), strxfrm(BA_LIB).

LEVEL

Level 1.

NAME

strerror – get error message string

SYNOPSIS

```
#include <string.h>
char *strerror (int errnum);
```

DESCRIPTION

The function `strerror()` maps the error number in *errnum* to an error message string, and returns a pointer to that string. `strerror()` uses the same set of error messages as `perror()`. The returned string should not be overwritten.

The message database `uxsyserr` is provided to make messages consistent. The messages for `strerror()` are obtained from this file via the System V messaging mechanism. Translated messages may be obtained by selecting the appropriate locale variables. [See `setlocale(BA_OS)`].

FILES

Message catalog: `uxsyserr`

SEE ALSO

perror(BA_LIB), setlocale(BA_OS).

LEVEL

Level 1.

NAME

strfmon – convert monetary value to string

SYNOPSIS

```
#include <monetary.h>

ssize_t *strfmon(char *s, size_t max, const char *format, . . .);
```

DESCRIPTION

strfmon is part of the X/Open Portability Guide Issue 4 optional Enhanced Internationalization feature group.

strfmon places characters into the array pointed to by **s** as controlled by the string pointed to by **format**. No more than **max** bytes are placed into the array.

format contains plain characters that are copied to the output stream, and conversion specifications, that result in the fetching of zero or more arguments which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are ignored.

A conversion specification consists of the following:

% character

optional flags

optional field width

optional precision

optional left precision

a conversion character that determines the conversion to be performed.

Options

The following flags can be specified to control the conversion:

=*f* An **=** followed by a single byte character **f** which is used as the numeric fill character. The default numeric fill character is the space character. This flag does not affect field width filling which always uses the space character. This flag is ignored unless a left precision is specified.

^ Do not format the currency amount with grouping characters. The default is to insert the grouping characters if defined for the current locale.

+ Specify the style of representing positive and negative amounts. You can only specify one of these. If **+** is specified, the locale's equivalent of **+** and **-** are used. If **(** is specified, negative amounts are enclosed within parentheses. **+** is the default.

! Suppress the currency symbol from the output conversion.

- Specify the alignment. If this flag is present all fields are left-justified rather than right-justified.

w A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified, or left-justified if the **-** flag is specified. The default is zero.

#n a **#** followed by a decimal digit string *n* specifying the maximum number of digits expect to be formatted to the left of the radix character. Use this option to keep the formatted output from multiple calls to the **strfmon** aligned in the same column. You can also use it to fill unused positions with a special character as in **$***123.45**. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character.

If grouping has not been suppressed with the ^ flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with space characters to make their positive and negative formats an equal length.

.p A period followed by a decimal digit string *p* specifying the number of digits after the radix character. If the value of the right precision *p* is zero, no radix character appears. If the right precision is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits before formatting.

The conversion characters and their meanings are:

i The *double* argument is formatted according to the locale's international currency format, for example, **USD 1,234.56** for the USA.

n The *double* argument is formatted according to the locale's national currency format, for example, **USD $1,234.56** for the USA.

% Convert to a **%**. No argument is converted. The entire conversion specification must be **%%**.

USAGE

The **LC_MONETARY** category of the program's locale affects the behavior of this function including the monetary radix character which may be different from the numeric radix character affected by this category. It also affects the grouping separator, the currency symbols, and formats. The international currency symbols used conform to **ISO 4217:1987** standard.

Return Values

If the total number of resulting bytes including the terminating null byte is not more than **maxsize**, **strfmon** returns the number of bytes placed into the array pointed to by **s**, not including the terminating null byte. Otherwise, **-1** is returned, the contents of the array is indeterminate, and **errno** is set to show the error.

Errors

In the following conditions, **strfmon** fails and sets **errno** to:

ENOSYS The function is not supported

E2BIG Conversion stopped because of lack of space in the buffer.

FUTURE DIRECTIONS

This interface will be mandatory in the future. Lowercase conversion characters are reserved for future use and uppercase for implementation- dependent use.

SEE ALSO

monetary(BA_OS)

LEVEL

Level 1.

NAME

strftime – convert date and time to string

SYNOPSIS

```
#include <time.h>

size_t strftime(char *s, size_t maxsize, const char *format,
    const struct tm *timeptr);
```

DESCRIPTION

strftime, places characters into the array pointed to by **s** as controlled by the string pointed to by *format*. The *format* string consists of zero or more directives and ordinary characters. All ordinary characters (including the terminating null character) are copied unchanged into the array. For **strftime**, no more than *maxsize* characters are placed into the array. For **strftime** the default format is the same as **"%c"**, for **cftime** and **ascftime** the default format is the same as **"%C"**. **cftime** and **ascftime** first try to use the value of the environment variable **CFTIME**, and if that is undefined or empty, the default format is used.

Each directive is replaced by appropriate characters as described by the following list. The appropriate characters are determined by the **LC_TIME** category of the program's locale and by the values contained in the structure pointed to by *timeptr* for **strftime**

%% same as %
%a abbreviated weekday name
%A full weekday name
%b abbreviated month name
%B full month name
%c basic date and time representation
%C number of the century (00 - 99)
%d day of month (01 - 31)
%D date as **%m/%d/%y**
%e day of month (1-31; single digits are preceded by a blank)
%h abbreviated month name.
%H hour (00 - 23)
%I hour (01 - 12)
%j day number of year (001 - 366)
%m month number (01 - 12)
%M minute (00 - 59)
%n same as new-line
%N date and time representation as used by **date**.
%p equivalent of either AM or PM
%r time in the a.m. and p.m. in the **C** locale it is equivalent to, **%I:%M:1P**)
%R same as **%H:%M**
%S seconds (00 - 61), allows for leap seconds
%t same as a tab
%T same as **%H:%M:%S**
%u weekday number (1 - 7), Monday = 1

%U week number of year (00 - 53), Sunday is the first day of week 1
%V week number of the year
%w weekday number (0 - 6), Sunday = 0
%W week number of year (00 - 53), Monday is the first day of week 1
%x locale's appropriate date representation
%X locale's appropriate time representation
%y year within century (00 - 99)
%Y year as ccyy (for example, 1986)
%Z time zone name or no characters if no time zone exists

The difference between **%U** and **%W** lies in which day is counted as the first of the week. Week number 01 is the first week in January starting with a Sunday for **%U** or a Monday for **%W**. Week number 00 contains those days before the first Sunday or Monday in January for **%U** and **%W**, respectively.

For **%V**, if the week containing January 1st has four or more days in the new year, it is week 1; otherwise, it is week 53 of the preceding year.

Modified Conversion Specifiers

O modifies the behavior of the following conversion specifiers. The decimal value is generated using the locale's alternate digit symbols.

%Od the day of the month, using alternative digit symbols filled as needed with leading zeros if available; otherwise, filled with spaces.
%Oe the day of the month, using alternative digit symbols filled with leading spaces as needed.
%OH the hour (24 hour clock), using alternative digit symbols.
%OI the hour (12 hour clock), using alternative digit symbols.
%Om the month using alternative digit symbols.
%OM the minutes using alternative digit symbols.
%OS the seconds using alternative digit symbols.
%Ou the weekday as a number using alternative digit symbols (Monday = 1).
%OU the week number using alternative digit symbols (see rules for **%U**).
%OV the week number using alternative digit symbols (see rules for **%V**).
%Ow the weekday as a number using alternative digit symbols (Sunday = 0).
%OW the week number using alternative digit symbols (see rules for **%W**).
%Oy the year (offset from **%C**) using alternative digit symbols.

E also modifies the behavior of the following conversion specifiers. An Era-specific value is generated instead of the normal value.ile.

%Ec Era-specific representation for date and time, as in **date**(1).
%EC Era-specific representation for the name of the base year (period).
%Ex Era-specific representation for the date.
%EX Era-specific representation for the time.
%Ey the offset from **%E** in the locale's alternative representation (year only).
%EY the full alternative year representation.

If the alternative format or specification for the above specifiers does not exist for the current locale, the behavior will be as if the unmodified specifier was used.

Selecting the Output's Language

By default, the output of **strftime**, appears as in the **C** locale. The user can request that the output of **strftime**, **cftime**, or **ascftime** be in a specific language by setting the *locale* for *category* **LC_TIME** in **setlocale**.

Timezone

The timezone is taken from the environment variable **TZ** [see **ctime**(BA_LIB) for a description of **TZ**].

Return Values

strftime returns the number of characters placed into the array pointed to by *s* not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate. If more than *maxsize* characters would have been placed into the array, **strftime** returns zero and the array content is indeterminate.

Files

LC_TIME file containing locale-specific date and time information

USAGE

The example illustrates the use of **strftime**. It shows what the string in **str** would look like if the structure pointed to by *tmptr* contains the values corresponding to Thursday, August 28, 1986 at 12:44:36 in New Jersey.

```
strftime(str, strsize, "%A %b %d %j", tmptr)
```

This results in **str** containing **Thursday Aug 28 240**, in the **C** locale.

For the following Era related definitions for **LC_TIME**:

```
era_d_fmt "%EY%mgatsu%dnichi (%a)"
era_d_fmt "The alternative time format is %h (%S) in %EC"
era_d_t_fmt "%EY%mgatsu%dnichi (%a) %T"
era "+:2:1990/01/01:+*:Heisei:%EC%Eynen";
      "+:1:1989/01/08:1989/12/31:Heisei:%ECgannen";
      "+:2:1927/01/01:1989/01/07:Shouwa:%EC%Eynen";
      "+:1:1926/12/25:1926/12/31:Shouwa:%ECgannen";
      "+:2:1913/01/01:1926/12/24:Taishou:%EC%Eynen";
      "+:1:1912/07/30:1912/12/31:Taishou:%ECgannen";
      "+:2:1869/01/01:1912/07/29:Meiji:%EC%Eynen";
      "+:1:1868/09/08:1868/12/31:Meiji:%ECgannen";
      "-:1868:1868/09/07:-*: :%Ey"
```

For August 1st 1912, with the **LC_TIME** locale category set as above:

```
strftime(str, strsize, "%Ey", tmptr);
```

would result in **str** containing **"01"**.

```
strftime(str, strsize, "%Ey %EC %Ex", tmptr);
```

would result in **str** containing **"Taishougannen Taishou Taishougannen08gatsu01nichi (Sun)"**.

```
strftime(str, strsize, "%EX", tmptr);
```

would result in **str** containing **"The alternative time format is Aug (01) in Taishou"**.

SEE ALSO

ctime(BA_LIB), **getenv**(BA_LIB)

LEVEL

Level 1.

NAME

string: **strcat**, **strncat**, **strcmp**, **strncmp**, **strcpy**, **strncpy**, **strdup**, **strlen**, **strchr**, **strrchr**, **strpbrk**, **strspn**, **strcspn**, **strtok**, **strstr** – string operations

SYNOPSIS

```
#include <string.h>

char *strcat(char *s1, const char *s2);

char *strncat(char *s1, const char *s2, size_t n);

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

char *strcpy(char *s1, const char *s2);

char *strncpy(char *s1, const char *s2, size_t n);

char *strdup(const char *s1);

size_t strlen(const char *s);

char *strchr(const char *s, int c);

char *strrchr(const char *s, int c);

char *strpbrk(const char *s1, const char *s2);

size_t strspn(const char *s1, const char *s2);

size_t strcspn(const char *s1, const char *s2);

char *strtok(char *s1, const char *s2);

char *strstr(const char *s1, const char *s2);
```

DESCRIPTION

The arguments *s*, *s1*, and *s2* point to strings (arrays of characters terminated by a null character). The functions **strcat**, **strncat**, **strcpy**, **strncpy**, and **strtok** alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

strcat appends a copy of string *s2*, including the terminating null character, to the end of string *s1*. **strncat** appends at most *n* characters. Each returns a pointer to the null-terminated result. The initial character of *s2* overrides the null character at the end of *s1*.

strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, based upon whether *s1* is lexicographically less than, equal to, or greater than *s2*. **strncmp** makes the same comparison but looks at most *n* characters. Characters following a null character are not compared.

strcpy copies string *s2* to *s1* including the terminating null character, stopping after the null character has been copied. **strncpy** copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

strdup returns a pointer to a new string which is a duplicate of the string pointed to by *s1*. The space for the new string is obtained using **malloc**(BA_OS). If the new string can not be created, a **NULL** pointer is returned.

strlen returns the number of characters in *s*, not including the terminating null character.

strchr (or **strrchr**) returns a pointer to the first (last) occurrence of *c* (converted to a **char**) in string *s*, or a **NULL** pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a **NULL** pointer if no character from *s2* exists in *s1*.

strspn (or **strcspn**) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a **NULL** pointer) will work through the string *s1* immediately following that token. In this way subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a **NULL** pointer is returned.

strstr locates the first occurrence in string *s1* of the sequence of characters (excluding the terminating null character) in string *s2*. **strstr** returns a pointer to the located string, or a null pointer if the string is not found. If *s2* points to a string with zero length (that is, the string **""**), the function returns *s1*.

SEE ALSO

malloc(BA_OS), **setlocale**(BA_OS), **strxfrm**(BA_LIB),

LEVEL

Level 1.

NOTICES

All of these functions assume the default locale ''C.'' For some locales, **strxfrm** should be applied to the strings before they are passed to the functions.

NAME

strptime – date and time conversion

SYNOPSIS

```
#include <time.h>

char *strptime(const char *buf, const char *format, struct tm *tm);
```

DESCRIPTION

strptime converts the character string pointed to by *buf* to values stored in the structure pointed to by *tm*, using the format specified by *format*.

format is composed of zero or more directives where each directive is composed of one of the following:

one or more white-space characters as specified by the **isspace** function,

an ordinary character (neither **%** or non white-space character), or

a conversion specification.

Conversion Specifications

Each conversion specification is composed of a **%** character followed by an optional modifier and then by a conversion character which specifies the replacement required. Usually, there should be white-space or other non-alphanumeric characters between any two conversion specifications. The following conversion specifications are supported:

%a locale's full or abbreviated weekday name
%A same as **%a**
%b locale's full or abbreviated month name
%B same as **%b**
%c locale's appropriate date and time representation (for example, **%x %X**)
%C number of the century (00 - 99), leading zeros are optional
%d day of month (01 - 31), leading zeros are optional
%D date as **%m/%d/%y**
%e same as **%d**
%h same as **%b**
%H hour (00 - 23), leading zeros are optional
%I hour (01 - 12), leading zeros are optional
%j day number of year (001 - 366), leading zeros are optional
%m month number (01 - 12), leading zeros are optional
%M minute (00 - 59), leading zeros are optional
%N date and time
%n any white space
%p locale's equivalent of either AM or PM
%r locale's time with 12-hour clock
%R time as **%H:%M**
%S seconds (00 - 61), allows for leap seconds, leading zeros are optional
%t any white space
%T time as **%H:%M:%S**
%U week number of year (00 - 53), Sunday is the first day of week 1, leading zeros are optional

%w weekday number (0 - 6), Sunday = 0, leading zeros are optional
%W week number of year (00 - 53), Monday is the first day of week 1, leading zeros are optional
%x locale's appropriate date representation
%X locale's appropriate time representation
%y year within century (00 - 99), leading zeros are optional
%Y year as ccyy (for example, 1986)
%% same as **%**

Modified Conversion Specifiers

Some directives can be modified by the **O** and **E** modifier characters to indicate that an alternative format or specification should be used instead of the normal directives. **%O** is the modifier used in association with the following conversion specifiers to specify that the locale's alternative digits be matched. The second letter has a similar effect as the letter excluding the **O** modifier.

%Od the day of the month, using the locale's alternative digit symbols filled as needed with leading zeros if available, otherwise, filled with spaces.
%Oe same as **%Od**
%OH the hour (24 hour clock), using the locale's alternative digit symbols.
%OI the hour (12 hour clock), using the locale's alternative digit symbols.
%Om the month using the locale's alternative digit symbols.
%OM the minutes using the locale's alternative digit symbols.
%OS the seconds using the locale's alternative digit symbols.
%OU the week number using the locale's alternative digit symbols (see rules for **%U**).
%Ow the weekday as a number using the locale's alternative digit symbols (Sunday = 0).
%OW the week number using the locale's alternative digit symbols (see rules for **%W**).
%Oy the year (offset from **%C**) using the locale's alternative digit symbols.

%E is a modifier used to match the date using different era information as specified in the **LC_TIME** locale data file.

%Ec the locale's alternative representation for date and time.
%EC the locale's alternative representation for the name of the base year (period).
%Ex the locale's alternative representation for the date.
%EX the locale's alternative representation for the time.
%Ey the offset from **%EC** in the locale's alternative representation (year only).
%EY the full alternative year representation.

A directive comprised of white-space characters is executed by scanning input up to the first character that is not white space which remains unscanned, or until no more characters can be scanned.

A directive that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of directives composed of **%n, %t**, white-space characters or any combination is executed by scanning up to the first character that is not white space which remains unscanned, or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, except the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate *tm* structure members are set to values corresponding to the locale information. Case is ignored when matching items are month or weekday names. If no match is found, **strptime** fails and no more characters are scanned.

Return Values

Upon successful completion, **strptime** returns a pointer to the character following the last character parsed. Otherwise, it returns a null pointer. If not implemented, **strptime** returns a null pointer and and sets **errno** to **ENOSYS**.

USAGE

Several ''**same as**'' format and the special processing of white-space characters are provided in order to ease the use of identical *format* strings for **strftime** and **strptime**.

SEE ALSO

strftime(BA_LIB), **time**(BA_ENV)

LEVEL

Level 1.

NAME

strtod, strtold, atof – convert string to double-precision number

SYNOPSIS

```
#include <stdlib.h>
double strtod(const char *str, char **ptr);
long double strtold(const char *str, char **ptr);
double atof(const char *str);
```

DESCRIPTION

The function `strtod()` returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

The function `strtod()` recognizes an optional string of white-space characters [as defined by `isspace()` in ctype(BA_LIB)], then an optional sign, then a string of digits optionally containing a decimal point character, then an optional exponent part consisting of an `e` or `E` followed by an optional sign, followed by one or more decimal digits.

If the value of *ptr* is not `(char **)0`, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, `*`*ptr* is set to *str*, and `0` is returned.

On the processors that support `strtold`, this function is equivalent to `strtod`, except that it returns a long double-precision floating-point number.

The function call `atof(`*str*`)` is equivalent to:

```
strtod(str, (char **)0)
```

RETURN VALUE

A macro `HUGE_VAL` will be defined by the `<math.h>` header file. This macro evaluates to a positive double expression, not necessarily representable as a float. On implementations that support the IEEE 754 standard, `HUGE_VAL` evalutates to $+\infty$.

If the correct value would cause overflow, $\pm$`HUGE_VAL` is returned (according to the sign of the value) and `errno` is set to `ERANGE`.

If the correct value would cause underflow, zero is returned and `errno` is set to `ERANGE`.

SEE ALSO

ctype(BA_LIB), scanf(BA_LIB), strtol(BA_LIB).

LEVEL

Level 1.

NAME

strtol, strtoul, atol, atoi – convert string to integer

SYNOPSIS

```
#include <stdlib.h>

long strtol(const char *str, char **ptr, int base);

unsigned long strtoul(const char *str, char **ptr,
     int base);

long atol(const char *str);

int atoi(const char *str);
```

DESCRIPTION

The function `strtol()` returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading white-space characters [as defined by `isspace()` in ctype(BA_LIB)] are ignored.

If the value of *ptr* is not `(char **)0`, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str* and zero is returned.

If *base* is positive (and not greater than `36`), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored and `0x` or `0X` is ignored if *base* is `16`.

If *base* is zero, the string itself determines the base in the following way: After an optional leading sign, a leading zero indicates octal conversion and a leading `0x` or `0X` hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from `long` to `int` can, of course, take place upon assignment or by an explicit cast.

`strtoul()` is similar to `strtol()` except that `strtoul()` returns as an unsigned long integer the value represented by *str*, and there can be no leading sign in *str*.

Except for the behavior on errors, the function call `atol(`*str*`)` is equivalent to:

```
strtol(str, (char **)0, 10)
```

Except for the behavior on errors, the function call `atoi(`*str*`)` is equivalent to:

```
(int)strtol(str, (char **)0, 10)
```

RETURN VALUE

If the argument *ptr* is a null pointer, the function `strtol()` will return the value of the string *str* as a long integer.

If the argument *ptr* is not `NULL`, the function `strtol()` will return the value of the string *str* as a long integer, and a pointer to the character terminating the scan will be returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to the argument *str* and the function `strtol()` returns `0`.

For `strtol()`, if the value represented by *str* would cause overflow, `LONG_MAX` or `LONG_MIN` is returned (according to the sign of the value), and `errno` is set to the value `ERANGE`.

For `strtoul()`, if the value represented by *str* would cause overflow, `ULONG_MAX` is returned, and `errno` is set to the value `ERANGE`.

SEE ALSO

ctype(BA_LIB), scanf(BA_LIB), strtod(BA_LIB).

LEVEL

Level 1.

NAME

strxfrm – string transformation

SYNOPSIS

```
#include <string.h>

size_t strxfrm(char *s1, const char *s2, size_t n);
```

DESCRIPTION

The function `strxfrm()` transforms the string *s2* and places the resulting string into the array *s1*. The transformation is such that if the `strcmp()` function [see string(BA_LIB)] is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the `strcoll()` function [see strcoll(BA_LIB)] applied to the same two original strings. The transformation is based on the program's locale for category `LC_COLLATE` [see setlocale(BA_OS)].

No more than *n* characters will be placed into the resulting array pointed to by *s1*, including the terminating null character. If *n* is zero, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The function `strxfrm()` returns the length of the transformed string (not including the terminating null character). If the value returned is *n* or more, the contents of the array *s1* are indeterminate.

USAGE

The transformation is such that two strings transformed by `strxfrm()` can be ordered by `memcmp()` or `strcmp()` and the results will be appropriate in terms of the collating sequence information in the program's locale.

EXAMPLE

The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by `s`.

```
1 + strxfrm((char *)NULL, s, 0);
```

SEE ALSO

memory(BA_LIB), setlocale(BA_OS), strcoll(BA_LIB), string(BA_LIB).

LEVEL

Level 1.

NAME

swab – swap bytes

SYNOPSIS

```
void swab (const char *from, char *to; int nbytes);
```

DESCRIPTION

The function `swab()` copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. This routine is useful for carrying binary data between machines with different low-order/high-order byte arrangements.

The argument *nbytes* should be even and non-negative. If the argument *nbytes* is odd and positive, the function `swab()` uses *nbytes–1* instead. If the argument *nbytes* is negative, the function `swab()` does nothing.

USAGE

Character movement is performed differently on different implementations; overlapping moves may yield unexpected results.

LEVEL

Level 1.

NAME

t_accept – accept a connect request

SYNOPSIS

```
#include <xti.h>
int t_accept(int fd, int resfd, struct t_call *call)
#include <tiuser.h>
int t_accept(int fd, int resfd, struct t_call *call)
```

Parameters

fd the file descriptor for the local transport endpoint where the connect request arrived.

resfd file descriptor for the local transport endpoint on which the connection is to be established.

call points to the **t_call** structure used to complete the connection.

DESCRIPTION

This function is one of the TLI/XTI routines used to establish a transport connection. It is invoked by an active transport user, following a call to **t_listen**, to accept a connection request from the transport interface and provide the information needed to complete a virtual connection.

It may also be used to pass a connection to another endpoint.

This function is a service of connection-mode transport providers and is supported only if the provider returned service type **T_COTS** or **T_COTS_ORD** on **t_open** or **t_getinfo**.

A transport user may accept a connection on either the same or local transport endpoint or on an endpoint different than the one on which the connect indication arrived. Before the connection can be accepted on the same endpoint (*resfd==fd*), the user must have responded to any previous connect indications received on that endpoint (via **t_accept** or **t_snddis**). Otherwise, **t_accept** will fail and set **t_errno** to **T_INDOUT**.

If a different transport endpoint is specified (*fs!=resfd*), then the user may or may not choose to bind the endpoint before **t_accept** is issued. If the endpoint is not bound, then the transport provider will automatically bind it to the same protocol address that **fd** is bound to. If the user chooses to bind to a local address, then *glen* must be zero for that protocol address, and the state of the endpoint must be **T_IDLE**. **t_accept** will change the address of *resfd* to be the same as that of *fd*. For portability, the first alternative is recommended.

Structure Definitions

The **t_call** structure contains the following members:

```
struct netbuf addr;      /* address          */
struct netbuf opt;       /* options          */
struct netbuf udata;     /* user data        */
int sequence;            /* sequence number  */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

In **t_call**, **addr** is the address of the caller, **opt** indicates any protocol-specific options associated with the connection, **udata** points to any user data to be returned to the caller, and **sequence** is the value returned by **t_listen** that uniquely associates the response with a previously received connect indication.

The values of parameters specified by **opt** and the syntax of those values are protocol specific. The **udata** argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned in the **connect** field of the **info** argument of **t_open** or **t_getinfo**. If the **len** field of **udata** is 0, no data will be sent to the caller.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint, or the user is invalidly accepting a connection on the same transport endpoint on which the connect indication arrived.

TOUTSTATE The function was issued in the wrong sequence on the transport endpoint referenced by *fd*, or the transport endpoint referred to by *resfd* is not in the **T_IDLE** state.

TACCES The user does not have permission to accept a connection on the responding transport endpoint or use the specified options.

TBADOPT The specified options were in an incorrect format or contained invalid information.

TBADDATA The amount of user data specified was not within the bounds supported by the transport provider as returned in the **connect** field of the **info** argument of **t_open** or **t_getinfo**.

TBADSEQ An invalid sequence number was specified.

TBADADDR The specified protocol address was in an incorrect format or contained illegal information.

TLOOK An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention. **t_accept** will fail and set **t_errno** to **TLOOK** when *fd* is not the same as *resfd* and there are indications (for example, a connect or disconnect) waiting to be received on that endpoint.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR	A system error has occurred during execution of this function.
TINDOUT	The function was called with *fd* equal to *resfd* but there are outstanding connection indications on the endpoint. The other connection indications must be handled either by rejecting them via **t_snddis** or accepting them via **t_accept**.
TPROVMISMATCH	The file descriptors *fd* and *resfd* do not refer to the same transport provider.
TRESQLEN	The endpoint referenced by *resfd* where *resfd!=fd* was bound to a protocol address with a **qlen** greater than 0.
TRESADDR	This transport provider requires that both *fd* and *resfd* be bound to the same address.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

fd	**T_INCON** on entry. **T_INCON**, **T_IDLE** or **T_DATAXFER** on exit.
resfd	**T_IDLE**, **T_UNBIND** on entry.

USAGE

When **t_accept** fails with a client timeout, this may be an indication that the client connection needs to be extended or that the server delay (between **t_listen** and **t_accept**) should be reduced.

A server application may retry **t_accept** unless a **TOUTSTATE** or **TSYSERR** error is received.

If the user does not specify protocol-specific options (the **len** field of *opt* is 0), it is assumed that the connection should accepted unconditionally. Options other than the defaults may be selected by the transport provider to ensure that the connection is accepted successfully.

SEE ALSO

t_connect(BA_LIB), **t_getinfo**(BA_LIB) **t_listen**(BA_LIB), **t_open**(BA_LIB), **t_rcvconnect**(BA_LIB), **t_snddis**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_alloc – allocate a data structure

SYNOPSIS

```
#include <xti.h>
char *t_alloc(int fd, int struct_type, int fields);
#include <tiuser.h>
char *t_alloc(int fd, int struct_type, int fields);
```

Parameters

fd the file descriptor for the transport endpoint.

struct_type identifies the type of structure for which memory should be allocated.

fields indicates fields for which buffers should be allocated.

DESCRIPTION

The **t_alloc** function is an TLI/XTI local management routine used to allocate data structures associated with the endpoint specified by *fd*. For *struct_type* **T_INFO**, *fd* is ignored, so that **T_INFO** structures may be allocated for use in calls to **t_open**.

t_alloc dynamically allocates memory for the various transport function argument structures as specified below. This function will allocate memory for the specified structure, and will also allocate memory for buffers referenced by the structure.

The structure to allocate is specified by **struct_type**, and can be one of the following:

```
T_BIND          /* struct t_bind          */
T_OPTMGMT       /* struct t_optmgmt       */
T_CALL          /* struct t_call          */
T_DIS           /* struct t_discon        */
T_UNITDATA      /* struct t_unitdata      */
T_UDERROR       /* struct t_uderr         */
T_INFO          /* struct t_info          */
```

where each of these structures may subsequently be used as an argument to one or more transport functions.

Structure Definitions

Each of the above structures, except **T_INFO**, contains at least one field of type **struct netbuf**. The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

For each field of this type, the user may specify that the buffer for that field should be allocated as well. The **fields** argument specifies this option, where the argument is the bitwise-OR of any of the following:

T_ADDR The **addr** field of the **t_bind**, **t_call**, **t_unitdata**, or **t_uderr** structures.

T_OPT The **opt** field of the **t_optmgmt**, **t_call**, **t_unitdata**, or **t_uderr** structures.

T_UDATA The **udata** field of the **t_call**, **t_discon**, or **t_unitdata** structures.

T_ALL All relevant fields of the given structure.

For each field specified in *fields*, **t_alloc** will allocate memory for the buffer associated with the field, initialize the **len** field to 0 and initialize the **buf** pointer and **maxlen** field accordingly. The length of the buffer allocated will be based on the same size information that is returned to the user on **t_open** and **t_getinfo**. Thus, *fd* must refer to the transport endpoint through which the newly allocated structure will be passed, so that the appropriate size information can be accessed.

If the size value associated with any specified field is –1, or –2, **t_alloc** will be unable to determine the size of the buffer to allocate and will fail with **t_errno** set to **TSYSERR**, unless when **T_ALL** is specified, in which case unsupported fields are ignored silently.

For any field not specified in *fields*, **buf** will be set to **NULL** and **maxlen** will be set to 0. If the *fields* argument is set to **T_ALL**, fields that are not supported by the transport provider specified by *fd* are not allocated.

Return Values

On successful completion, **t_alloc** returns a pointer to the newly allocated structure. On failure, **NULL** is returned, and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TSYSERR A system error has occurred during execution of this function.

TNOSTRUCTYPE The argument that specifies *struct_type* is invalid, for example, because the type of structure requested is inconsistent with the transport provider (connection mode or connectionless) indicated by *fd*.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_alloc has no effect on state. Valid states are **T_UNBND, T_IDLE, T_OUTCON, T_INCON, T_DATAXFER, T_OUTREL** and **T_INREL** on entry. On exit, they are unchanged.

USAGE

Use of **t_alloc** to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface.

Buffers and memory that have been allocated with **t_alloc** may be freed with **t_free**.

SEE ALSO

t_free(BA_LIB), **t_getinfo**(BA_LIB), **t_open**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_bind – bind an address to a transport endpoint

SYNOPSIS

```
#include <xti.h>
int t_bind(int fd, struct t_bind *req, struct t_bind *ret)
#include <tiuser.h>
int t_bind(int fd, struct t_bind *req, struct t_bind *ret)
```

Parameters

fd the file descriptor for the transport endpoint

req points to the **t_bind** structure used to identify the request.

ret points to the **t_bind** structure used to identify the return.

DESCRIPTION

This function is an TLI/XTI local management routine that associates a protocol address with the transport endpoint specified by *fd* and activates the endpoint.

If *fd* refers to a connection-mode service, the transport provider may then begin listening for connect indications on that endpoint (**t_listen**), or the provider may begin sending connection requests from that transport endpoint (**t_connect**).

If *fd* refers to a connectionless service, the transport user may then proceed with sending or receiving data units through the transport endpoint (**t_snd, t_rcv**).

Structure Definitions

The *req* and *ret* arguments point to a **t_bind** structure containing the following members:

```
struct netbuf addr;   /* address                */
unsigned qlen;        /* connect indications    */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char       *buf;
```

len specifies the number of bytes in the address, **buf** points to the address buffer, and **maxlen** is the maximum size of the address buffer. The **qlen** field, in connection mode only, is used to indicate the maximum number of outstanding connect indications.

In *req*, **len** and **buf** are used to specify the protocol address to be bound to the transport endpoint. **maxlen** has no meaning for the *req* argument.

In *ret*, the user specifies **maxlen** (which is the maximum size of the address buffer) and **buf** (which points to the buffer where the address is to be placed).

On return, *ret* contains the bound address. This is the same as the address specified by the user in *req*. **len** specifies the number of bytes in the bound address and **buf** points to the bound address. If **maxlen** is not large enough to hold the returned address, an error will result. If the requested address is not available, **t_bind** fails with an error and **t_errno** is set to **TADDRBUSY**.

If no address is specified in *req* (the **len** field in **addr** is 0 or *req* is NULL), the transport provider will assign an appropriate address to be bound, and will return that address in *ret*.

req may be **NULL** if the user does not want to specify the protocol address to be bound. Here, the value of **qlen** is assumed to be zero, and the transport provider must assign an address to the transport endpoint. Similarly, *ret* may be **NULL** if the user does not care what address was bound by the provider and is not interested in the negotiated value of **qlen**.

It is also valid to set *req* and *ret* to **NULL** for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

The **qlen** field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider. A value of **qlen** greater than 0 is only meaningful when issued by a passive transport user that expects other users to call it. The value of **qlen** will be negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, the **qlen** field in *ret* will contain the negotiated value.

Return Values

t_bind returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TOUTSTATE The function was issued in the wrong sequence.

TBADADDR The specified protocol address was in an incorrect format or contained illegal information.

TNOADDR The transport provider could not allocate an address.

TACCES The user does not have permission to use the specified address.

TBUFOVFLW The number of bytes (**maxlen**) allocated for an incoming argument is greater than zero but not sufficient to store the value of that argument. The provider's state will change to **T_IDLE** and the information to be returned in *ret* will be discarded.

TSYSERR A system error has occurred during execution of this function.

TADDRBUSY In connection mode, the requested address has already been bound to another transport endpoint.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, **T_UNBND**; **T_IDLE** on exit.

USAGE

The following notes are for connection-mode service.

This function allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must support this capability also), but it is not allowable to bind more than one protocol address to the same transport endpoint.

If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address., In other words, only one **t_bind** for a given protocol address may specify a value of **gen** greater than 0. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication.

If a user attempts to bind a protocol address to a second transport endpoint with a value of **glen** greater than 0, **t_bind** will fail with **TADDRBUSY**.

A transport provider may not allow an explicit binding of more than one endpoint to the same protocol address, although it allows more than one connection to be recommended not to bind transport endpoints that are used as responding endpoints (*resfd*) in a call to **t_accept**, if the responding address is to be the same as the called address.

If a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of that connection. No other transport endpoints may be bound for listening while that initial listening endpoint is in the data transfer phase. This will prevent more than one transport endpoint bound to the same protocol address from accepting connection indications.

Warnings

Note that the behavior of **t_bind** has changed in order to conform to X/OPEN's TLI/XTI specifications. Previously, if *req* was specified **t_bind** returned an alternate address if the one requested was busy. Now, **t_bind** will fail and **t_error** will be set to **TADDRBUSY**. Thus now, in case of failure, applications need to check the value of **e_errno** and repeat the call with a different address if the one requested is busy (or not requested a specific address). Also, applications need not verify the address they were bound to if they requested an address and **t_bind** succeeded.

SEE ALSO

t_alloc(BA_LIB), **t_connect**(BA_LIB), **t_listen**(BA_LIB), **t_open**(BA_LIB), **t_unbind**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_close – close a transport endpoint

SYNOPSIS

```
#include <xti.h>
int t_close(int fd);
#include <tiuser.h>
int t_close(int fd);
```

Parameters

fd the file descriptor for the transport endpoint specified by *fd*.

DESCRIPTION

This function is an TLI/XTI local management routine used to close a transport endpoint. The **t_close** function indicates to the transport provider that the user is finished with the transport endpoint specified by *fd*. In addition, **t_close** closes the file associated with the transport endpoint and frees any local library resources associated with the endpoint.

Return Values

t_close returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, any except **T_UNINIT**; **T_UNINIT** on exit.

USAGE

t_close should be called from the **T_UNBND** state. However, this function does not check state information, so it may be called from any valid state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically.

Warnings

If **t_close** is issued while a transport address is bound to an endpoint, the address will be unbound.

If **t_close** is called when the transport connection is still active, the connection will be aborted, the file descriptor will be closed, and the transport connection associated with that endpoint will be broken for any process that references that endpoint.

t_close should not be issued on a connection endpoint before data has been successfully transmitted and received or data may be lost.

SEE ALSO

t_getstate(BA_LIB) **t_open**(BA_LIB), **t_unbind**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_connect – establish a connection with another transport user

SYNOPSIS

```
#include <xti.h>
int t_connect(int fd, struct t_call *sndcall struct t_call *rcvcall)
#include <tiuser.h>
int t_connect(int fd, struct t_call *sndcall struct t_call *rcvcall)
```

Parameters

fd the file descriptor for the transport endpoint where the connection will be established.

sndcall points to the **t_call** structure used to identify the transport user sending the connection indication.

rcvcall points to the **t_call** structure used to identify the transport user that will receive the connection indication.

DESCRIPTION

This TLI/XTI routine enables a transport user to request a connection to the specified destination transport user.

This function is a service of connection-mode transport providers and is supported only if the provider returned service type **T_COTS** or **T_COTS_ORD** on **t_open** or **t_getinfo**.

sndcall specifies information needed by the transport provider to establish a connection and *rcvcall* specifies information that is associated with the newly established connection.

Structure Definitions

The pointers *sndcall* and *rcvcall* refer to a **t_call** structure that contains the following members:

```
struct netbuf addr;   /* address          */
struct netbuf opt;    /* options          */
struct netbuf udata;  /* user data        */
int sequence;         /* sequence number  */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

In *sndcall*, **addr** specifies the protocol address of the destination transport user, **opt** presents any protocol-specific information that might be needed by the transport provider, **udata** points to optional user data that may be passed to the destination transport user during connection establishment, and **sequence** has no meaning for this function.

On return in *rcvcall*, **addr** returns the protocol address associated with the responding transport endpoint, **opt** presents any protocol-specific information associated with the connection, **udata** points to optional user data that may be returned by the destination transport user during connection establishment, and **sequence** has no

meaning for this function.

The **opt** argument implies no structure on the options that may be passed to the transport provider. The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user may choose not to negotiate protocol options by setting the **len** field of **opt** to 0. In this case, the provider may use default options.

The **udata** argument enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned in the **connect** field of the **info** argument of **t_open** or **t_getinfo**. If the **len** field of **udata** is 0 in *sndcall*, no data will be sent to the destination transport user.

On return, the **addr**, **opt**, and **udata** fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the **maxlen** field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *rcvcall* may be NULL, in which case no information is given to the user on return from **t_connect**.

Return Values

t_connect returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TOUTSTATE The function was issued in the wrong sequence.

TNODATA **O_NONBLOCK** was set (by **t_open** or **cntl**) so the function executed in asynchronous mode. Therefore, the connection establishment procedure was successfully executed, but the function did not wait for a response from the remote user.

TBADADDR The specified protocol address was in an incorrect format or contained invalid information.

TBADOPT The specified protocol options were in an incorrect format or contained invalid information.

TBADDATA The amount of user data specified was not within the bounds supported by the transport provider as returned in the **connect** field of the **info** argument of **t_open** or **t_getinfo**.

TACCES

TLOOK	An asynchronous event has occurred on the transport endpoint specified by *fd* and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution of this function.
TADDRBUSY	The specified connection already exists, and this transport user does not support multiple connections with the same pair of local and remote addresses.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, **T_IDLE**; **T_OUTCON** or **TDATAXFER** (successful) or **T_IDLE** (failed) on exit. If **t_connect** fails with a **TLOOK** or **TNODATA** error, a change of state may occur.

USAGE

By default, **t_connect** executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (that is, return value of 0) indicates that the requested connection has been established. However, if **O_NONBLOCK** is set (via **t_open** or **fcntl**), **t_connect** executes in asynchronous mode. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user, and may fail with **t_error** set to **TNODATA**.

Also, in the case of the TCP protocol, the peer TCP, and not the peer transport user, confirms the connection. One consequence of this fact is that the **t_connect** can return success, even though the remote server process may (later) call **t_snddis**, rather than **t_accept**, thus aborting the connection.

SEE ALSO

t_accept(BA_LIB), **t_getinfo**(BA_LIB) **t_listen**(BA_LIB), **t_open**(BA_LIB), **t_optmgmt**(BA_LIB), **t_rcvconnect**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_error – write an error message

SYNOPSIS

```
#include <xti.h>

int t_error(char *errmsg);

extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;

#include <tiuser.h>

int t_error(char *errmsg);
```

Parameters

errmsg a user-supplied error message that gives context to the error.

t_errno index to a user-specified message array.

t_errlist points to the array of user-supplied message strings.

t_nerr maximum number of messages in the user-specified message array.

DESCRIPTION

This function is an TLI/XTI local management routine used to generate a message under error conditions. **t_error** writes a message on the standard error output describing the last error encountered during a call to a transport function.

The argument string *errmsg* is user supplied and may be set to give context to the error. The message returned by **t_error** prints in the following format: the user-supplied error message followed by a colon and the standard transport function error message for the current value contained in **t_errno**.

t_errlist and **t_nerr** are maintained for compatibility and should not be used. In their place use **t_strerror**(BA_LIB).

Return Values

Upon completion, a value of 0 is returned. No errors are defined.

State Transitions

t_error may be issued from any valid state except **T_UNINIT** and has no effect on the entry state at exit.

USAGE

On return, **t_errno** is set when an error occurs and is not cleared on subsequent successful calls.

If the returned value of **t_errno** has been set to **TSYSERR**, **t_error** will also print the standard error message for the current value contained in **errno**

Examples

Following a **t_connect** function call, which might fail on a transport endpoint *fd2* because a bad address was detected, a call to **t_error** might be issued to check for a possible failure:

```
t_error("t_connect failed on fd2");
```

If the **t_connect** fails, **t_errno** is set to the appropriate value, and the diagnostic message would print as:

```
t_connect failed on fd2:  Incorrect transport address format
```

where "**t_connect failed on fd2**" tells the user which function failed on which transport endpoint, and "**Incorrect transport address format**" identifies the specific error that occurred.

SEE ALSO

pfmt(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

t_errlist and **t_nerr** are Level 2, effective January 1995.

NAME

t_free – free a data structure

SYNOPSIS

```
#include <xti.h>
int t_free(char *ptr, int struct_type);
#include <tiuser.h>
int t_free(char *ptr, int struct_type);
```

Parameters

ptr points to the structure referenced by **t_alloc**.

struct_type identifies the type of structure.

DESCRIPTION

The **t_free** function frees memory previously allocated by **t_alloc**. This function will free memory for the specified structure, and will also free memory for buffers referenced by the structure.

ptr points to the structure, previously referenced by **t_alloc**, which may be one of six types described by *struct_type*. One of the following types of structures may be specified:

```
T_BIND          /* struct t_bind          */
T_OPTMGMT       /* struct t_optmgmt       */
T_CALL          /* struct t_call          */
T_DIS           /* struct t_discon        */
T_UNITDATA      /* struct t_unitdata      */
T_UDERROR       /* struct t_uderr         */
T_INFO          /* struct t_info          */
```

where each of these structures is used as an argument to one or more transport functions.

t_free will check the **addr**, **opt**, and **udata** fields of the given structure (as appropriate), and free the buffers pointed to by the **buf** field of the **netbuf** structure. If **buf** is **NULL**, **t_free** will not attempt to free memory. After all buffers are freed, **t_free** will free the memory associated with the structure pointed to by *ptr*.

Undefined results will occur if *ptr* or any of the **buf** pointers points to a block of memory that was not previously allocated by **t_alloc**.

Return Values

t_free returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to the following:

TSYSERR A system error has occurred during execution of this function.

TNOSTRUCTYPE The argument that specifies *struct_type* is invalid, for example, because the type of structure requested in inconsistent with the transport provider (connection mode or connectionless).

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_free may be issued from any valid state except **T_UNINIT** and has no effect on the entry state at exit.

USAGE

After all buffers are freed, **t_free** will free the memory associated with the structure pointed to by *ptr*.

If **buf** is **NULL**, **t_free** will not attempt to free memory.

Warnings

Undefined results will occur if *ptr* or any of the **buf** pointers points to a block of memory that was not previously allocated by **t_alloc**.

SEE ALSO

t_alloc(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_getinfo – get protocol-specific service information

SYNOPSIS

```
#include <xti.h>
int t_getinfo(int fd, struct t_info *info);
#include <tiuser.h>
int t_getinfo(int fd, struct t_info *info);
```

Parameters

fd the file descriptor for the transport endpoint

info points to the **t_info** structure used to identify a transport provider.

DESCRIPTION

This function is an TLI/XTI local management routine used to return the current characteristics of the underlying transport protocol associated with file descriptor *fd*. The **t_info** structure is used to return the same information returned by **t_open**. This function enables a transport user to access this information during any phase of communication.

Structure Definitions

This argument points to a **struct t_info** which contains the following members:

```
long addr;      /* max size of the transport protocol address           */
long options;   /* max num of bytes of protocol-specific options        */
long tsdu;      /* max size of a transport service data unit (TSDU)     */
long etsdu;     /* max size of an expedited TSDU (ETSDU)                */
long connect;   /* max amt of data allowed on connect establishment     */
long discon;    /* max amt of data allowed on t_snddis, t_rcvdis        */
long servtype;  /* service type supported by transport provider         */
long flags;     /* provides more info about transport provider          */
```

The values of the fields have the following meanings:

addr A value greater than or equal to 0 indicates the maximum size of a transport protocol address, and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to 0 indicates the maximum number of bytes of protocol-specific options supported by the provider, and a value of –2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than 0 specifies the maximum size of a transport service data unit (**TSDU**); a value of 0 specifies that the transport provider does not support the concept of **TSDU**, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a **TSDU**, and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than 0 specifies the maximum size of an expedited transport service data unit (**ETSDU**); a value of 0 specifies that the transport provider does not support the concept of **ETSDU**, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an **ETSDU**, and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than 0 specifies the maximum amount of data that may be associated with connection establishment functions; and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than 0 specifies the maximum amount of data that may be associated with the **t_snddis** and **t_rcvdis** functions, and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider. A single transport endpoint may support only one of the following services at one time.

- **T_COTS** The transport provider supports a connection-mode service but does not support the optional orderly release facility.
- **T_COTS_ORD** The transport provider supports a connection-mode service with the optional orderly release facility.
- **T_CLTS** The transport provider supports a connectionless service. For this service type, **t_open** will return –2 for **etsdu**, **connect**, and **discon**.

flags This field specifies other information in the form of bit indicators as follows: If **T_SENDZERO** is on, this indicates that the underlying transport provider supports the sending of 0-length TSDUs.

Return Values

t_getinfo returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to the following:

TSYSERR A system error has occurred during execution of this function.

TBADF The specified file descriptor does not refer to a transport endpoint.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_getinfo may be issued from any valid state except **T_UNINIT** and has no effect on the entry state at exit.

USAGE

If a transport user is concerned with protocol independence, the sizes specified in **t_info** may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the **t_alloc** function may be used to allocate these buffers.

The value of each field may change as a result of protocol option negotiation during connection establishment. These values will only change from the values presented to **t_open** after the endpoint enters the **T_DATAFER** state.

Warnings

An error will result if the data size allowed is exceeded by the transport user on any function.

SEE ALSO

t_alloc(BA_LIB), **t_close**(BA_LIB) **t_open**(BA_LIB),

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_getprotaddr – get protocol addresses

SYNOPSIS

```
#include <xti.h>

int t_getprotaddr(int fd, struct t_bind *boundaddr,
      struct t_bind *peeraddr);
```

Parameters

fd the file descriptor for the transport endpoint associated with the protocol address.

boundaddr points to the bound address of the local transport endpoint.

peeraddr points to the peer address.

DESCRIPTION

This function is an TLI/XTI local management function used to get protocol addresses for both the local and remote endpoints. **t_getprotaddr** returns, for the transport endpoint specified by *fd*, the local address of the transport endpoint (pointed to by *boundaddr*) and the remote address of the peer (pointed to by *peeraddr*).

The local address is available if the endpoint is bound (not in the **T_UNBND** state) and the peer address is available if the endpoint is in the **T_DATAXFER** state.

Structure Definitions

boundaddr and *peeraddr* point to a **t_bind** structure containing the following members:

```
struct netbuf addr;   /* address                */
unsigned qlen;        /* connect indications    */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

len specifies the number of bytes in the address, **buf** points to the address buffer, and **maxlen** is the maximum size of the address buffer. The **qlen** field, in connection mode only, is used to indicate the maximum number of outstanding connect indications.

In *boundaddr* and *peeraddr*, the **maxlen** field is the maximum size of the address buffer, specified by the user, and **buf** points to the buffer where the address will be placed.

On return, if the endpoint specified by *fd* is currently bound, the **buf** field of *boundaddr* points to the address of the transport endpoint and the **len** field indicates the length of the address. If the endpoint is not bound, the **len** field of *boundaddr* returns a value of 0.

If the transport user is in the **T_DATAXFER** state, the **buf** field of *peeraddr* points to the address of the peer (currently connected to *fd* and the **len** field indicates the length of that address. If the endpoint is not connected, the **len** field of *peeraddr* returns a value of 0.

Return Values

t_getprotaddr returns a value of 0 on successful completion and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The number of bytes (**maxlen**) allocated for an incoming argument is greater than zero but not sufficient to store the value of that argument.
TSYSERR	A system error has occurred during execution of this function.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_getprotaddr may be issued from any valid state except **T_UNINIT** and has no effect on the entry state at exit.

USAGE

This function is applicable for both connection-mode and connectionless transport services. However, since the remote endpoint is never in the **TDATAXFER** state if the service is connectionless, only the address of the bound endpoint will be returned.

SEE ALSO

t_accept(BA_LIB), **t_bind**(BA_LIB), **t_connect**(BA_LIB)

LEVEL

Level 1.

NAME

t_getstate – get the current state

SYNOPSIS

```
#include <xti.h>
int t_getstate(int fd);
#include <tiuser.h>
int t_getstate(int fd);
```

Parameters

fd the file descriptor for the transport endpoint associated with the current state.

DESCRIPTION

This function is an TLI/XTI local management routine used to return the current state of the provider associated with the transport endpoint specified by *fd*.

TLI/XTI states are changed by user events that reflect the success or failure of calls to the various TLI/XTI functions. Because fewer TLI/XTI user events occur over connectionless services, there are fewer TLI/XTI states than for connection-mode services.

The current state may be one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending (connection mode only)
T_INCON	incoming connection pending (connection mode only)
T_DATAXFER	data transfer (connection mode only)
T_OUTREL	outgoing orderly release (waiting for an orderly release indication) (connection mode only)
T_INREL	incoming orderly release (waiting for an orderly release request) (connection mode only)

Return Values

t_getstate returns the current state on successful completion and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSTATECHNG	The transport provider is undergoing a state change.
TSYSERR	A system error has occurred during execution of this function.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_getstate may be issued from any valid state except **T_UNINIT** and has no effect on the entry state.

USAGE

The **t_getstate** function is applicable to both connection-mode and connectionless transport services.

Warnings

If the provider is undergoing a state transition when **t_getstate** is called, the function will fail.

SEE ALSO

t_getinfo(BA_LIB), **t_open**(BA_LIB),

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_listen – listen for a connect request

SYNOPSIS

```
#include <xti.h>
int t_listen(int fd, struct t_call *call);
#include <tiuser.h>
int t_listen(int fd, struct t_call *call);
```

Parameters

fd the file descriptor for the transport endpoint where connect indications arrive.

call points to the **t_call** structure used to describe the connect indications.

DESCRIPTION

This function is an TLI/XTI routine for use in establishing a transport connection. **t_listen** listens for a connect request from a calling transport user and is designed for use by server applications using connection-mode transport services.

fd identifies the local transport endpoint where connect indications arrive, and on return, *call* contains information describing the connect indication.

Structure Definitions

call points to a **t_call** structure, which contains the following members:

```
struct netbuf addr;      /* address          */
struct netbuf opt;       /* options          */
struct netbuf udata;     /* user data        */
int sequence;            /* sequence number  */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

In *call*, **addr** returns the protocol address of the calling transport user, **opt** returns protocol-specific parameters associated with the connect request, **udata** returns any user data sent by the caller on the connect request, and **sequence** is a number that uniquely identifies the returned connect indication. The value of **sequence** enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the **addr**, **opt**, and **udata** fields of *call*, the **maxlen** field of each must be set before issuing **t_listen** to indicate the maximum size of the buffer for each.

Return Values

t_listen returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TBADQLEN The argument **qlen** of the endpoint specified by *fd* is 0.

TBUFOVFLW The number of bytes (**maxlen**) allocated for an incoming argument is greater than zero but not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to **T_INCON**, and the connect indication information to be returned in **call** is discarded.

TNODATA **O_NONBLOCK** was set, but no connect indications had been queued.

TLOOK An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR A system error has occurred during execution of this function.

TQFULL The maximum number of connect indications has been reached for the endpoint specified by *fd*.

TOUTSTATE The function was issued in the wrong sequence on the transport endpoint referenced by **fd**.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

T_IDLE on entry. **T_INCON** (successful) or **T_IDLE** (no requests) on exit.

SEE ALSO

t_accept(BA_LIB), **t_bind**(BA_LIB), **t_connect**(BA_LIB), **t_open**(BA_LIB), **t_rcvconnect**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_look – check for asynchronous event

SYNOPSIS

```
#include <xti.h>
int t_look(int fd);
#include <tiuser.h>
int t_look(int fd);
```

Parameters

fd the file descriptor for the local transport endpoint associated with the current event.

DESCRIPTION

This function is an TLI/XTI local management routine used to return the current asynchronous event on the transport endpoint specified by *fd*. The event indicated reflects the service type of the transport provider. **t_look** enables a transport provider to notify a transport user, when the user is issuing functions in synchronous mode, if an asynchronous event has occurred on the specified endpoint.

Certain events require immediate notification of the user and are indicated by a specific error, **TLOOK**, on the current or next function to be executed.

This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Values returned by **t_look** include the following:

T_LISTEN A request for a connection (connect indication) has arrived at the transport endpoint.

T_CONNECT A connect confirmation (confirmation of connect indication) has arrived at the transport endpoint. (When the server accepts a connect request, the confirmation is generated.)

T_DATA User data has arrived at the transport endpoint.

T_EXDATA Expedited user data has arrived at the transport endpoint.

T_DISCONNECT A notification that the connection was aborted or that the server did not accept a connect request (disconnect indication) has arrived at the transport endpoint.

T_UDERR Notification that a datagram error occurred (unitdata error indication) has arrived at the transport endpoint.

T_ORDREL A request for the orderly release of a connection (orderly release indication) has arrived at the transport endpoint.

T_GODATA Notification that it is again possible to send user data has arrived at the transport endpoint.

T_GOEXDATA Notification that it is again possible to send expedited user data has arrived at the transport endpoint.

Return Values

On success, **t_look** returns 0 if no event exists or the value that indicates which event exists. On failure, –1 is returned and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSYSERR	A system error has occurred during execution of this function.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_look may be issued from any valid state except **T_UNINIT** and has no effect on the state.

SEE ALSO

t_open(BA_LIB), **t_snd**(BA_LIB) **t_sndudata**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_open – establish a transport endpoint

SYNOPSIS

```
#include <xti.h>
#include <fcntl.h>
int t_open(const char *path, int oflag, struct t_info *info)
#include <tiuser.h>
#include <fcntl.h>
int t_open(const char *path, int oflag, struct t_info *info)
```

Parameters

path points to the path name of the file to open.

oflag identifies any open flags. *oflag* may be constructed from **O_NONBLOCK OR**-ed with **O_RDWR**. These flags are defined in the header file **<fcntl.h>**.

info points to the **t_info** structure used to identify a transport provider.

DESCRIPTION

The **t_open** function is an TLI/XTI local management routine that must be called as the first step in the initialization of a transport endpoint. This function opens a UNIX file that identifies a transport endpoint connected to a chosen transport provider (that is, transport protocol). The file descriptor (*fd*) for the opened file identifies the provider and establishes the endpoint. For example, a call to **t_open** may be used to open the file **/dev/iso_cots** to specify an OSI connection-oriented transport layer protocol as the transport provider.

The file descriptor returned by **t_open** is be used by all subsequent functions to identify the particular local transport endpoint.

t_open also returns various default characteristics of the underlying transport protocol by setting fields in the **t_info** structure.

Structure Definitions

This argument points to a **struct t_info** which contains the following members:

```
long addr;      /* max size of the transport protocol address          */
long options;   /* max num of bytes of protocol-specific options       */
long tsdu;      /* max size of a transport service data unit (TSDU)    */
long etsdu;     /* max size of an expedited TSDU (ETSDU)               */
long connect;   /* max amt of data allowed on connect establishment    */
long discon;    /* max amt of data allowed on t_snddis, t_rcvdis       */
long servtype;  /* service type supported by transport provider        */
long flags;     /* provides more info about transport provider         */
```

The values of the fields have the following meanings:

addr A value greater than or equal to 0 indicates the maximum size of a transport protocol address, and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to 0 indicates the maximum number of bytes of protocol-specific options supported by the provider, and a value of –2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than 0 specifies the maximum size of a transport service data unit (**TSDU**); a value of 0 specifies that the transport provider does not support the concept of **TSDU**, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a **TSDU**, and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than 0 specifies the maximum size of an expedited transport service data unit (**ETSDU**); a value of 0 specifies that the transport provider does not support the concept of **ETSDU**, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an **ETSDU**, and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to 0 specifies the maximum amount of data that may be associated with connection establishment functions, and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to 0 specifies the maximum amount of data that may be associated with the **t_snddis** and **t_rcvdis** functions, and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider. A single transport endpoint may support only one of the following services at one time.

- **T_COTS** The transport provider supports a connection-mode service but does not support the optional orderly release facility.
- **T_COTS_ORD** The transport provider supports a connection-mode service with the optional orderly release facility.
- **T_CLTS** The transport provider supports a connectionless service. For this service type, **t_open** will return –2 for **etsdu**, **connect**, and **discon**.

flags This bit field is used to specify other information about the transport provider. If the **T_SENDZERO** bit is set in **flags**, this indicates the underlying transport provider supports the sending of zero-length TSDUs.

A single transport endpoint may support only one of the above services at one time.

If *info* is set to **NULL** by the transport user, no protocol information is returned by **t_open**.

Return Values

t_open returns a valid file descriptor on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to the following:

TSYSERR	A system error has occurred during execution of this function.
TBADFLAG	An invalid flag is specified.
TBADNAME	An invalid path is specified for the transport provider name.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, **T_UNINIT**; **T_UNBND** (successful) or **T_UNINIT** (failed) on exit.

USAGE

If a transport user is concerned with protocol independence, the sizes specified in **t_info** may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the **t_alloc** function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

If **info** is set to **NULL** by the transport user, no protocol information is returned by **t_open**.

Warnings

If **t_open** is used on a non-XTI-conforming STREAMS device, unpredictable events may occur.

The **close**() system call should not be used directly on the file descriptor returned by **t_open**(BA_LIB). The t_close(BA_LIB) routine should be used to close a file descriptor opened by **t_open**(BA_LIB).

SEE ALSO

t_alloc(BA_LIB), **t_close**(BA_LIB), **t_getinfo**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_optmgmt – manage options for a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_optmgmt (int fd, struct t_optmgmt *req, struct t_optmgmt *ret);
```

Parameters

fd the file descriptor for the transport endpoint

req points to the **t_optmgmt** structure used to identify the request.

info points to the **t_optmgmt** structure used to identify the return.

DESCRIPTION

The **t_optmgmt** function enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider associated with the bound transport endpoint specified by *fd*. **t_optmgmt** is a TLI local management routine that may be used with both connection-mode and connectionless protocol services.

Structure Definitions

The *req* and *ret* arguments point to a **t_optmgmt** structure containing the following members:

```
struct netbuf opt;    /* protocol options   */
long flags;           /* actions            */
```

The **opt** field identifies protocol options and the **flags** field is used to specify the action to take with those options.

The options are represented by a **netbuf** structure in a manner similar to the address used in **t_bind**. The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

req is used to request a specific action of the provider and to send options to the provider. **len** specifies the number of bytes in the options, **buf** points to the options buffer, and **maxlen** has no meaning for the *req* argument.

The transport provider may return options and flag values to the user through *ret*. For *ret*, **maxlen** specifies the maximum size of the options buffer and **buf** points to the buffer where the options are to be placed. On return, **len** specifies the number of bytes of options returned. **maxlen** has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold.

The actual structure and content of the options is imposed by the transport provider.

The **flags** field of *req* can specify one of the following actions:

T_NEGOTIATE This action enables the user to negotiate the values of the options specified in *req* with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through *ret*.

T_CHECK This action enables the user to verify whether the options specified in *req* are supported by the transport provider. On return, the **flags** field of *ret* will have either **T_SUCCESS** or **T_FAILURE** set to indicate to the user whether the options are supported. These flags are only meaningful for the **T_CHECK** request.

T_DEFAULT This action enables a user to retrieve the default options supported by the transport provider into the **opt** field of *ret*. In *req*, the **len** field of **opt** must be zero and the **buf** field may be **NULL**.

Return Values

t_optmgmt returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TOUTSTATE The function was issued in the wrong sequence.

TACCES The user does not have permission to negotiate the specified options.

TBADOPT The specified protocol options were in an incorrect format or contained illegal information.

TBADFLAG An invalid flag was specified.

TBUFOVFLW The number of bytes (**maxlen**) allocated for an incoming argument is greater than zero but not sufficient to store the value of that argument. The information to be returned in **ret** will be discarded.

TSYSERR A system error has occurred during execution of this function.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

TNOTSUPPORT The action is not supported by the transport provider.

State Transitions

t_optmgmt may be issued from any valid state except **T_UNINIT** and has no effect on the state.

USAGE

If issued as part of a connectionless service, **t_optmgmt** may block due to flow control constraints. The function will not complete until the transport provider has processed all previously sent data units.

Warnings

The transport provider interface may not support the functionality for **T_NEGOTIATE** and/or **T_CHECK**, causing **t_optmgmt** to fail with a **TNOTSUPPORT** error.

SEE ALSO

t_accept(BA_LIB), **t_alloc**(BA_LIB), **t_bind**(BA_LIB), **t_connect**(BA_LIB), **t_getinfo**(BA_LIB), **t_listen**(BA_LIB), **t_open**(BA_LIB), **t_rcvconnect**(BA_LIB)

FUTURE DIRECTIONS

To allow conformance to X/Open Transport Interface (XTI), **t_optmgmt** will be modified to support XPG4 options management. Application writers and protocol providers must be aware of this migration due to the incompatibilities it will produce. In addition, the inclusion of the header **tiuser.h** has been moved to Level 2 to accommodate this migration from TLI routines to XTI routines.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_rcv – receive normal or expedited data sent over a connection

SYNOPSIS

```
#include <xti.h>
int t_rcv(int fd, char *buf, unsigned int nbytes, int *flags);
#include <tiuser.h>
int t_rcv(int fd, char *buf, unsigned int nbytes, int *flags);
```

Parameters

fd the file descriptor for the transport endpoint through which data will arrive.

buf points to the receive buffer where user data will be placed.

nbytes specifies the size of the receive buffer.

flags specifies optional flags on return.

DESCRIPTION

This function is an TLI/XTI connection-mode data transfer routine which is issued to notify a transport user that there is normal or expedited data to be received over a connection. The messages sent to the transport user may be 0-length.

By default, **t_rcv** operates in synchronous mode and will wait for data to arrive if none is currently available. However, if **O_NONBLOCK** is set (via **t_open** or **fcntl**), **t_rcv** will execute in asynchronous mode and will fail if no data is available. (See **TNODATA** below.)

On return from the call, if **T_MORE** is set in **flags**, this indicates that there is more data and the current transport service data unit (**TSDU**) or expedited transport service data unit (**ETSDU**) must be received in multiple **t_rcv** calls.

Each **t_rcv** with the **T_MORE** flag set indicates that another **t_rcv** must follow to get more data for the current **TSDU**. The end of the **TSDU** is identified by the return of a **t_rcv** call with the **T_MORE** flag not set.

If the transport provider does not support the concept of a **TSDU** as indicated in the *info* argument on return from **t_open** or **t_getinfo**, the **T_MORE** flag is not meaningful and will be ignored.

On return from the call, if **T_EXPEDITED** is set in **flags** the data returned is expedited data. If the number of bytes of expedited data exceeds *nbytes*, **t_rcv** will set **T_EXPEDITED** and **T_MORE** on return from the initial call. Subsequent calls to retrieve the remaining **ETSDU** will have **T_EXPEDITED** set on return. The end of the **ETSDU** is identified by the return of a **t_rcv** call with the **T_MORE** flag not set.

If expedited data arrives after part of a **TSDU** has been retrieved, receipt of the remainder of the **TSDU** will be suspended until the **ETSDU** has been processed. Only after the full **ETSDU** has been retrieved (**T_MORE** not set) will the remainder of the **TSDU** be available to the user.

Return Values

On successful completion, **t_rcv** returns the number of bytes received. On failure, it returns –1 and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TNODATA	**O_NONBLOCK** was set, but no data is currently available from the transport provider.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution of this function.
TOUTSTATE	The function was issued in the wrong sequence on the transport endpoint referenced by **fd**.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, **T_DATAXFER** or **T_OUTREL**; unchanged (successful) on exit.

USAGE

t_rcv is applicable only for connection-mode transport services.

In synchronous mode, **t_look** may alternatively be used to notify the transport user that normal or expedited data has been received or that flow control restrictions have been lifted. Additional functionality is provided by the Event Management Interface.

SEE ALSO

t_getinfo(BA_LIB), **t_look**(BA_LIB), **t_open**(BA_LIB), **t_snd**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_rcvconnect – receive the confirmation from a connect request

SYNOPSIS

```
#include <xti.h>

int t_rcvconnect(int fd, struct t_call *call)

#include <tiuser.h>

int t_rcvconnect(int fd, struct t_call *call)
```

Parameters

fd the file descriptor for the transport endpoint where communication will be established.

call points to the **t_call** structure used to identify the transport user that will receive the connection indication.

DESCRIPTION

t_rcvconnect enables a calling transport user to determine the status of a connect request that it issued to a responding transport endpoint. On successful completion of **t_rcvconnect**, the connection is established.

By default, **t_rcvconnect** executes in synchronous mode and waits for the connection to be established before returning. In asynchronous mode, this function is used in conjunction with **t_connect** to establish a connection.

fd identifies the responding transport endpoint, and *call* contains information associated with the newly established connection.

Structure Definitions

The *call* argument points to a **t_call** structure which contains the following members:

```
struct netbuf addr;      /* address          */
struct netbuf opt;       /* options          */
struct netbuf udata;     /* user data        */
int sequence;            /* sequence number  */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

In *call*, **addr** returns the protocol address associated with the responding transport endpoint, **opt** presents any protocol-specific information associated with the connection, **udata** points to optional user data that may be returned by the destination transport user during connection establishment, and **sequence** has no meaning for this function.

The **maxlen** field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *call* may be **NULL**, in which case no information is given to the user on return from **t_rcvconnect**.

On return, the **addr**, **opt**, and **udata** fields reflect values associated with the connection.

If **O_NONBLOCK** is set (via **t_open** or **fcntl**), **t_rcvconnect** executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, **t_rcvconnect** fails on a **TNODATA** error and returns immediately without waiting for the connection to be established.

Return Values

t_rcvconnect returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW The number of bytes (**maxlen**) allocated for an incoming argument is greater than zero but not sufficient to store the value of that argument. The connect information to be returned in **call** will be discarded. The provider's state, as seen by the user, will be changed to **DATAXFER**.

TNODATA **O_NONBLOCK** was set, but a connect confirmation has not yet arrived.

TLOOK An asynchronous event has occurred on the transport connection specified by *fd* and requires immediate attention.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR A system error has occurred during execution of this function.

TOUTSTATE The function was issued in the wrong sequence on the transport endpoint referenced by **fd**.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, **T_OUTCON**; **T_DATAXFER** (successful) or **T_OUTCON** (failed) on exit.

USAGE

A subsequent call to **t_rcvconnect** is required to complete the connection establishment phase and retrieve the information returned in *call*.

SEE ALSO

t_accept(BA_LIB), **t_bind**(BA_LIB), **t_connect**(BA_LIB), **t_listen**(BA_LIB), **t_open**(BA_LIB), **t_optmgmt**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **`tiuser.h`** is Level 2 effective January 1995.

NAME

t_rcvdis – retrieve information from disconnect

SYNOPSIS

```
#include <xti.h>
int t_rcvdis(int fd, struct t_discon *discon);
#include <tiuser.h>
int t_rcvdis(int fd, struct t_discon *discon);
```

Parameters

fd the file descriptor for the transport endpoint where the connection had been established.

discon points to the **t_discon** structure associated with the disconnect information.

DESCRIPTION

This function is an TLI/XTI connection release routine used to identify the cause of a disconnect and to retrieve any user data sent with the disconnect.

fd is used by the calling transport user to identify the local transport endpoint where the connection existed, and *discon* points to a **t_discon** structure associated with the disconnection.

Structure Definitions

The *discon* argument points to a **t_discon** structure containing the following members:

```
struct netbuf udata;  /* user data    */
int reason;           /* reason code  */
int sequence;         /* connect ind. */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

reason specifies the reason for the disconnect through a protocol-dependent reason code, **udata** identifies any user data that was sent with the disconnect, and **sequence** may identify an outstanding connect indication with which the disconnect is associated. **sequence** is only meaningful when **t_rcvdis** is issued by a passive transport user who has executed one or more **t_listen** functions and is processing the resulting connect indications.

If a disconnect indication occurs, **sequence** can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of **reason** or **sequence**, **discon** may be **NULL**, and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (via **t_listen**) and **discon** is **NULL**, the user will be unable to identify which connect indication the disconnect is associated with.

Return Values

t_rcvdis returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TNODIS No disconnect indication currently exists on the specified transport endpoint.

TBUFOVFLW The number of bytes (**maxlen**) allocated for an incoming argument is greater than zero but not sufficient to store the value of that argument. The provider's state, as seen by the user, will change to **T_IDLE**, and the disconnect indication information to be returned in **discon** will be discarded.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR A system error has occurred during execution of this function.

TOUTSTATE The function was issued in the wrong sequence on the transport endpoint referenced by *fd*, or the transport endpoint referred to by *resfd* is not in the **T_IDLE** state.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_rcvdis may be issued from any valid state except **T_UNINIT**, **T_UNBND**, or **T_IDLE**. Valid states on exit are **T_IDLE** (successful) and **T_INCON** (successful but there are connect indications outstanding).

SEE ALSO

t_connect(BA_LIB), **t_listen**(BA_LIB), **t_open**(BA_LIB), **t_snddis**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_rcvrel – acknowledge receipt of an orderly release indication

SYNOPSIS

```
#include <xti.h>
int t_rcvrel(int fd);
#include <tiuser.h>
int t_rcvrel(int fd);
```

Parameters

fd the file descriptor for the transport endpoint where the connect indication is received.

DESCRIPTION

This function is an TLI/XTI connection release routine used to acknowledge receipt of an orderly release indication. In **t_rcvrel**, *fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user should not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if **t_sndrel** has not been issued by the user.

This function is an optional service of the transport provider, and is only supported if the transport provider returned service type **T_COTS_ORD** on **t_open** or **t_getinfo**.

Return Values

t_rcvrel returns 0 on success and –1 on failure **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TNOREL No orderly release indication currently exists on the specified transport endpoint.

TLOOK An asynchronous event has occurred on the transport endpoint specified by *fd* and requires immediate attention.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR A system error has occurred during execution of this function.

TOUTSTATE The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

T_DATAXFER on entry and **T_INREL** on exit; or **T_OUTREL** on entry and **T_IDLE** on exit.

SEE ALSO

t_open(BA_LIB), **t_sndrel**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_rcvudata – receive a data unit

SYNOPSIS

```
#include <xti.h>
int t_rcvudata(int fd, struct t_unitdata *unitdata, int *flags);
#include <tiuser.h>
int t_rcvudata(int fd, struct t_unitdata *unitdata, int *flags);
```

Parameters

fd the file descriptor for the transport endpoint through which the data will be received.

unitdata points to the **t_unitdata** structure associated with the received data unit.

flags points to a value set on return if the complete data unit was not received.

DESCRIPTION

This function is an TLI/XTI connection release routine used in connectionless mode to receive a data unit from another transport user. Data is received through the transport endpoint specified by *fd* and *unitdata* points to information associated with the data unit.

On return, *flags* points to a value that indicates whether the complete data unit was received.

This function is a service of connectionless transport providers and is supported only if the provider returned service type **T_CLTS** on **t_open** or **t_getinfo**.

Structure Definitions

The *unitdata* argument points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;   /* address      */
struct netbuf opt;    /* options      */
struct netbuf udata;  /* user data    */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

The **maxlen** field of **addr**, **opt**, and **udata** must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, **addr** specifies the protocol address of the sending user, **opt** identifies protocol-specific options that were associated with this data unit, and **udata** specifies the user data that was received.

If the buffer defined in the **udata** field of **unitdata** is not large enough to hold the current data unit, the buffer will be filled and **T_MORE** will be set in **flags** on return to indicate that another **t_rcvudata** should be issued to retrieve the rest of the data unit. Subsequent **t_rcvudata** call(s) will return 0 for the length of the address and options until the full data unit has been received.

Return Values

t_rcvudata returns 0 on successful completion and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TNODATA **O_NONBLOCK** was set, but no data units are currently available from the transport provider.

TBUFOVFLW The number of bytes (**maxlen**) allocated for an incoming argument is greater than zero but not sufficient to store the value of that argument. The unit data information to be returned in **unitdata** will be discarded.

TLOOK An asynchronous event has occurred on the transport endpoint specified by *fd* and requires immediate attention.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TOUTSTATE The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

TSYSERR A system error has occurred during execution of this function.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, **T_IDLE**; unchanged on exit.

USAGE

By default, **t_rcvudata** operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if **O_NONBLOCK** is set (via **t_open** or **fcntl**), **t_rcvudata** will execute in asynchronous mode and will fail if no data units are available.

SEE ALSO

t_getinfo(BA_LIB), **t_open**(BA_LIB), **t_rcvuderr**(BA_LIB), **t_sndudata**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_rcvuderr – receive a unit data error indication

SYNOPSIS

```
#include <xti.h>

int t_rcvuderr(int fd, struct t_uderr *uderr);

#include <tiuser.h>

int t_rcvuderr(int fd, struct t_uderr *uderr);
```

DESCRIPTION

The function `t_rcvuderr()` is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. *fd* identifies the local transport endpoint through which the error report will be received, and *uderr* points to a `t_uderr` structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
long error;
```

The `maxlen` field of `addr` and `opt` must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, the `addr` structure specifies the destination protocol address of the erroneous data unit, the `opt` structure identifies protocol-specific options that were associated with the data unit, and `error` specifies a protocol-dependent error code.

If the user does not care to identify the data unit that produced an error, *uderr* may be set to `NULL`, and `t_rcvuderr()` will simply clear the error indication without reporting any information to the user.

RETURN VALUE

Upon successful completion, the function `t_rcvuderr()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `t_errno` to indicate an error.

ERRORS

Under the following conditions, the function `t_rcvuderr()` fails and sets `t_errno` to:

`TBADF` if the specified file descriptor does not refer to a transport endpoint.

`TNOUDERR` if no unit data error indication currently exists on the specified transport endpoint.

`TBUFOVFLW` if the number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error information to be returned in *uderr* will be discarded.

TNOTSUPPORT	if this function is not supported by the underlying transport provider.
TSYSERR	if a system error has occurred during execution of this function.

SEE ALSO

t_look(BA_LIB), t_rcvudata(BA_LIB), t_sndudata(BA_LIB).

FUTURE DIRECTIONS

The inclusion of the header `tiuser.h` has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace `tiuser.h` with `xti.h`.

LEVEL

Level 1. The inclusion of the header `tiuser.h` is Level 2 effective January 1995.

NAME

t_snd – send normal or expedited data over a connection

SYNOPSIS

```
#include <xti.h>
int t_snd(int fd, void *buf, unsigned int nbytes, int flags);
#include <tiuser.h>
int t_snd(int fd, void *buf, unsigned int nbytes, int flags);
```

Parameters

fd the file descriptor for the transport endpoint over which data will be sent.

buf points to the user data.

nbytes specifies the number of bytes of user data to be sent.

flags specifies optional flags on return.

DESCRIPTION

This function is an TLI/XTI data transfer routine used to send either normal or expedited data over a connection.

By default, **t_snd** operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if **O_NONBLOCK** is set (via **t_open** or **fcntl**), **t_snd** will execute in asynchronous mode, and will fail immediately if there are flow control restrictions.

Even when there are no flow control restrictions, **t_snd** will wait if **STREAMS** internal resources are not available, regardless of the state of **O_NONBLOCK**.

On successful completion, **t_snd** returns the number of bytes accepted by the transport provider. Normally this will equal the number of bytes specified in *nbytes*. However, if **O_NONBLOCK** is set, it is possible that only part of the data will be accepted by the transport provider. In this case, **t_snd** will set **T_MORE** for the data that was sent (see below) and will return a value less than *nbytes*. If *nbytes* is 0 and the sending of 0 bytes is not supported by the underlying transport provider, **t_snd** will return –1 with **t_errno** set to **TBADDATA**. A return value of 0 indicates that the request to send a 0-length data message was sent to the provider.

If **T_EXPEDITED** is set in *flags*, the data will be sent as expedited data, and will be subject to the interpretations of the transport provider.

If **T_MORE** is set in *flags*, or is set as described above, an indication is sent to the transport provider that the transport service data unit (**TSDU**) or expedited transport service data unit (**ETSDU**) is being sent through multiple **t_snd** calls. Each **t_snd** with the **T_MORE** flag set indicates that another **t_snd** will follow with more data for the current **TSDU**. The end of the **TSDU** (or **ETSDU**) is identified by a **t_snd** call with the **T_MORE** flag not set. Use of **T_MORE** enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a **TSDU** as indicated in the *info* argument on return from **t_open** or **t_getinfo**, the **T_MORE** flag is not meaningful and should be ignored.

The size of each **TSDU** or **ETSDU** must not exceed the limits of the transport provider as returned by **t_open** or **t_getinfo**. If the size is exceeded, a **TSYSERR** with system error **EPROTO** will occur. However, the **t_snd** may not fail because **EPROTO** errors may not be reported immediately. In this case, a subsequent call that accesses the transport endpoint will fail with the associated **TSYSERR**.

Return Values

On successful completion, **t_snd** returns the number of bytes accepted by the transport provider. On failure, it returns –1 and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TFLOW **O_NONBLOCK** was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR A system error has been detected during execution of this function.

TBADDATA *nbytes* is 0 and sending 0 bytes is not supported by the transport provider; or, the number of bytes on a single send was greater than the number specified for *nbytes* by the *info* argument on the **t_open** or **fcntl**; or, the maximum size was exceeded during multiple sends.

TLOOK An asynchronous event has occurred on the transport endpoint specified by *fd* and requires immediate attention.

TBADFLAG An invalid flag was specified.

TOUTSTATE The function was issued in the wrong sequence on the transport endpoint referenced by **fd**.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, **T_DATAXFER** or **T_INREL**; unchanged on exit.

USAGE

t_snd is applicable only for connection-mode transport services that return a service type of **T_COTS** or **T_COTS_ORD** in response to **t_open** or **t_getinfo**.

Warnings

The **t_snd** routine does not look for a disconnect indication (showing that the connection was broken) before passing data to the provider.

In asynchronous mode, if the number of bytes accepted exceeds the number requested by the transport provider, the provider may be blocked because of flow control.

If several processes issue concurrent calls to **t_snd** (multiple sends), the data from those processes may be intermixed (since several users of the same endpoint are treated as a single user by the transport provider).

If the maximum size of a **TSDU** or **ETSDU** is exceeded as a result of multiple sends, XTI may not detect the error. If the error is detected, **t_snd** fails with **TBADDATA**. If the error is not detected, **t_snd** or a subsequent call fails on an error indicating that the connection has been aborted.

SEE ALSO

fcntl(BA_OS), **poll**(BA_OS), **t_getinfo**(BA_LIB), **t_look**(BA_LIB), **t_open**(BA_LIB), **t_rcv**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_snddis – send user-initiated disconnect request

SYNOPSIS

```
#include <xti.h>
int t_snddis(int fd, struct t_call *call);
#include <tiuser.h>
int t_snddis(int fd, struct t_call *call);
```

Parameters

fd the file descriptor for the transport endpoint where the connection exits.

call points to the **t_call** structure associated with information about the connection.

DESCRIPTION

This function is issued by a transport user to initiate a release on an already established connection with a responding transport endpoint, specified by *fd*. It may also be issued to to reject a connect request.

The values pointed to by *call* have different semantics that vary with the context of the call.

This function is a service of connection-mode transport providers and is supported only if the provider returned service type **T_COTS** or **T_COTS_ORD** on **t_open** or **t_getinfo**.

Structure Definitions

The *call* argument points to a **t_call** structure that contains the following members:

```
struct netbuf addr;      /* address          */
struct netbuf opt;       /* options          */
struct netbuf udata;     /* user data        */
int sequence;            /* sequence number  */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

When rejecting a connect request, **call** must be non-**NULL** and contain a valid value of **sequence** to identify uniquely the rejected connect indication to the transport provider. The **addr** and **opt** fields of **call** are ignored.

In all other cases, **call** need only be used when data is being sent with the disconnect request. The **addr**, **opt**, and **sequence** fields of the **t_call** structure are ignored. If the user does not want to send data to the remote user, the value of **call** may be **NULL**.

udata specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned in the **discon** field of the *info* argument of **t_open** or **t_getinfo**. If the **len** field of **udata** is zero, no data will be sent to the remote user.

Return Values

t_snddis returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TOUTSTATE The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.

TBADDATA The amount of user data specified was not within the bounds supported by the transport provider as returned in the **discon** field of the **info** argument of **t_open** or **t_getinfo**. The transport provider's outgoing queue will be flushed, so data may be lost.

TBADSEQ An invalid sequence number was specified, or a **NULL** call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost.

TLOOK An asynchronous event has occurred on the transport endpoint specified by *fd* and requires immediate attention.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR A system error has occurred during execution of this function.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_snddis may be issued from any valid state except **T_UNINIT**, **T_UNBND**, or **T_IDLE**. Valid states on exit are **T_IDLE** (successful) and **T_INCON** (successful but there are connect indications outstanding).

USAGE

After issuing t_snddis, the user may not send any more data over the connection. However, a user may continue to receive data if a disconnect request has not been received (see **t_rcvdis**).

Warnings

When executed, **t_snddis** causes an abortive disconnect, which may result in a loss of data sent by **t_snd** but not yet received. The return of an error does not preclude loss of data.

SEE ALSO

t_connect(BA_LIB), **t_getinfo**(BA_LIB), **t_listen**(BA_LIB), **t_open**(BA_LIB), **t_rcvdis**(BA_LIB), **t_rcvrel**(BA_LIB), **t_snd**(BA_LIB), **t_sndrel**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_sndrel – initiate an orderly release

SYNOPSIS

```
#include <xti.h>
int t_sndrel(int fd);
#include <tiuser.h>
int t_sndrel(int fd);
```

Parameters

fd the file descriptor for the transport endpoint where the connection exists.

DESCRIPTION

This function is an TLI/XTI connection release routine used to initiate an orderly release of a transport connection associated with the transport endpoint specified by *fd*. **t_sndrel** indicates to the transport provider that the transport user has no more data to send.

This function is an optional service of the transport provider and is only supported if the transport provider returned service type **T_COTS** or **T_COTS_ORD** on **t_open** or **t_getinfo**.

Return Values

t_sndrel returns 0 on success and –1 on failure and **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	**O_NONBLOCK** was set, but the flow control mechanism prevented the transport provider from accepting the function at this time.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution of this function.
TLOOK	An asynchronous even has occurred on the transport endpoint referenced by *fd* and requires immediate attention.
TOUTSTATE	The function was issued in the wrong sequence on the transport endpoint referenced by **fd**.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

T_DATAXFER on entry and **T_OUTREL** on exit; or **T_INREL** on entry and **T_IDLE** on exit.

USAGE

After issuing **t_sndrel**, the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has not been received.

If **t_sndrel** is issued from an invalid state, the provider will generate an **EPROTO** protocol error; however, this error may not occur until a subsequent reference to the transport endpoint.

SEE ALSO

t_open(BA_LIB), **t_rcvrel**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_sndudata – send a data unit

SYNOPSIS

```
#include <xti.h>
int t_sndudata(int fd, struct t_unitdata *unitdata);
#include <tiuser.h>
int t_sndudata(int fd, struct t_unitdata *unitdata);
```

Parameters

fd the file descriptor for the transport endpoint through which data will be sent.

unitdata points to the **t_unitdata** structure associated with the transmitted data unit.

DESCRIPTION

This function is used in connectionless mode to send a data unit to another transport user. Data is sent through the transport endpoint specified by *fd*, which must be bound, and *unitdata* points to information associated with the data unit.

This function is a service of connectionless mode transport providers and is supported only if the provider returned service type **T_CLTS** on **t_open** or **t_getinfo**.

Structure Definitions

The *unitdata* argument points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;   /* address      */
struct netbuf opt;    /* options      */
struct netbuf udata;  /* user data    */
```

The **netbuf** structure contains the following members:

```
unsigned int    maxlen;
unsigned int    len;
char            *buf;
```

In *unitdata*, **addr** specifies the protocol address of the destination user, **opt** identifies protocol-specific options that the user wants associated with this request, and **udata** specifies the user data to be sent. The user may choose not to specify what protocol options are associated with the transfer by setting the **len** field of **opt** to 0. In this case, the provider may use default options.

If the **len** field of **udata** is 0, and the sending of 0 bytes is not supported by the underlying transport provider, **t_sndudata** will return –1 with **t_errno** set to **TBADDATA**.

Return Values

t_sndudata returns 0 on successful completion and –1 on failure **t_errno** is set to indicate the error.

Errors

On failure, **t_errno** may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	**O_NONBLOCK** was set, but the flow control mechanism prevented the transport provider from accepting data at this time.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution of this function. (An **EPROTO** error may not cause **t_sndudata** to fail until subsequent access of the transport endpoint.)
TBADDATA	**nbytes** is 0 and sending 0 bytes is not supported by the transport provider.
TLOOK	An asynchronous event has occurred on the transport endpoint specified by *fd* and requires immediate attention.
TBADADDR	The specified protocol address was in an incorrect format or contained invalid information. (This error may alternatively be returned by **t_rcvuderr**.)
TBADOPT	The specified protocol options were in an incorrect format or contained invalid information. (This error may alternatively be returned by **t_rcvuderr**.)
TOUTSTATE	The function was issued in the wrong sequence on the transport endpoint referenced by **fd**.
TPROTO	A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

On entry, **T_IDLE**; unchanged on exit.

USAGE

By default, **t_sndudata** operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if **O_NONBLOCK** is set (via **t_open** or **fcntl**), **t_sndudata** will execute in asynchronous mode and will fail under such conditions.

The calling process can use **t_look** or the Event Management Interface to determine when flow control restrictions, if any, have been cleared.

Warnings

If **t_sndudata** is issued before the destination user has activated its transport endpoint (see **t_bind**), the data unit may be discarded.

If **t_sndudata** is issued from an invalid state, or if the amount of data specified in **udata** exceeds the **TSDU** size as returned in the **tsdu** field of the *info* argument of **t_open** or **t_getinfo**, the provider will generate an **EPROTO** protocol error. If the state is invalid, this error may not occur until a subsequent reference is made to the transport endpoint.

If a unit data error is received, a subsequent call should be made to **t_rcvuderr** to check for conditions indicated by **TBADADDR** and **TBADOPT**, which are not always returned by **t_sndudata**.

SEE ALSO

t_bind(BA_LIB), **t_getinfo**(BA_LIB), **t_open**(BA_LIB), **t_rcvudata**(BA_LIB), **t_rcvuderr**(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_strerror – get error message string

SYNOPSIS

```
#include <xti.h>

char *t_strerror(int errnum);
```

Parameters

errnum the TLI/XTI number for the language-dependent error message string.

DESCRIPTION

The **t_strerror** function is an TLI/XTI local management routine that returns, for the error number specified by *errnum*, the pointer to a language dependent error message string.

When **t_strerror** is issued, the contents of the string pointed to on return are not modified, but may be modified by a subsequent call to **t_strerror**.

The comments used in the header file **xti.h** to describe the values in **t_errno** are identical to the error message string pointed to by **t_strerror** on return. If the language is not English, the text provided is equivalent.

The error message string itself is not ended by a newline character.

If the value supplied in *errnum* is not recognized, the response from **t_strerror** is a pointer to the following string:

```
<errnum>: error unknown
```

where **<errnum>** is the value supplied on the call.

Return Values

t_strerror returns a string pointer to the requested error. No errors are defined.

State Transitions

t_strerror may be issued from any valid state except **T_UNINIT** and has no effect on the entry state at exit.

SEE ALSO

t_error(BA_LIB),

LEVEL

Level 1.

NAME

t_sync – synchronize transport library

SYNOPSIS

```
#include <xti.h>
int t_sync(int fd);
#include <tiuser.h>
int t_sync(int fd);
```

Parameters

fd the file descriptor for the transport endpoint for which data structures will be synchronized.

DESCRIPTION

This function is an TLI / XTI local management routine used to synchronize XTI data structures and protocol specific information. For the transport endpoint specified by *fd*, **t_sync** synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert a raw file descriptor to an initialized transport endpoint, assuming that the file descriptor referenced a transport provider.

This function also allows two cooperating processes to synchronize their interaction with a transport provider. For example, if a process **forks** a new process and issues an **exec**, the new process must issue a **t_sync** to build the private library data structure associated with a transport endpoint and to synchronize the data

T_INREL incoming orderly release (waiting for an orderly release request) (connection mode only)

Errors

On failure, **t_errno** may be set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TSTATECHNG The transport provider is undergoing a state change.

TSYSERR A system error has occurred during execution of this function.

TPROTO A communication problem has been detected with the transport provider and there is no other value of **t_errno** to describe the error condition.

State Transitions

t_sync may be issued from any valid state except **T_UNINIT** and has no effect on the entry state at exit.

USAGE

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, those activities should be coordinated so as not to violate the state of the provider.

Warnings

If the transport endpoint specified by *fd* is undergoing a state transition when **t_sync** is called, the function will fail.

SEE ALSO

t_getstate(BA_LIB)

FUTURE DIRECTIONS

The inclusion of the header **tiuser.h** has been moved to Level 2 due to the migration from TLI routines to the X/Open XTI routines. Replace **tiuser.h** with **xti.h**.

LEVEL

Level 1.

The inclusion of the header **tiuser.h** is Level 2 effective January 1995.

NAME

t_unbind – disable a transport endpoint

SYNOPSIS

```
#include <xti.h>

int t_unbind(int fd);
```

DESCRIPTION

The function `t_unbind()` disables the transport endpoint specified by *fd* which was previously bound by `t_bind()` [see t_bind(BA_LIB)]. On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

RETURN VALUE

Upon successful completion, the function `t_unbind()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `t_errno` to indicate an error.

ERRORS

Under the following conditions, the function `t_unbind()` fails and sets `t_errno` to:

`TBADF`	if the specified file descriptor does not refer to a transport endpoint.
`TOUTSTATE`	if the function was issued in the wrong sequence.
`TLOOK`	if an asynchronous event has occurred on this transport endpoint.
`TSYSERR`	if a system error has occurred during execution of this function.

SEE ALSO

t_bind(BA_LIB).

LEVEL

Level 1.

NAME

tmpfile – create a temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void);
```

DESCRIPTION

The function `tmpfile()` creates a temporary file using a name generated by the `tmpnam()` routine [see tmpnam(BA_LIB)], and returns a corresponding pointer to the `FILE` structure associated with the stream. The temporary file will automatically be deleted when the process that opened it terminates or the temporary file is closed. The temporary file is opened for update (`w+`) [see fopen(BA_OS)].

RETURN VALUE

If the temporary file cannot be opened, a `NULL` pointer is returned.

ERRORS

Under the following conditions, the function `tmpfile()` fails and sets `errno` to:

`EMFILE` if `{OPEN_MAX}` file descriptors are currently open in the calling process.

`ENFILE` if the system file table is full.

`ENOSPC` if the directory or file system that would contain the new file cannot be expanded.

SEE ALSO

creat(BA_OS), fopen(BA_OS), mktemp(BA_LIB), tmpnam(BA_LIB), unlink(BA_OS).

LEVEL

Level 1.

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam(char *s);

char *tempnam(const char *dir, const char *pfx);
```

DESCRIPTION

These functions generate filenames that can safely be used for a temporary file.

The function `tmpnam()` always generates a filename using the path-prefix defined by the `<stdio.h>` header file as `P_tmpdir`. If the argument *s* is `NULL`, the function `tmpnam()` leaves its result in an internal static area and returns a pointer to that area. The next call to the function `tmpnam()` will destroy the contents of the area. If the argument *s* is not `NULL`, it is assumed to be the address of an array of at least `L_tmpnam` bytes, where `L_tmpnam` is a constant defined by the `<stdio.h>` header file; the function `tmpnam()` places its result in that array and returns *s*.

The function `tempnam()` allows the user to control the choice of a directory. If defined in the user's environment, the value of the environmental variable `TMPDIR` is used as the name of the desired temporary file directory. The argument *dir* points to the name of the directory in which the file is to be created. If the argument *dir* is `NULL` or points to a string that is not a name for an appropriate directory, the path-prefix defined by the `<stdio.h>` header file as `P_tmpdir` is used. If that directory is not accessible, the directory `/tmp` will be used.

The function `tempnam()` uses the `malloc()` routine [see malloc(BA_OS)] to get space for the constructed filename, and returns a pointer to this area. Thus, any pointer value returned from the function `tempnam()` may serve as an argument to the function `free()` [see `free()` in malloc(BA_OS)]. If the function `tempnam()` cannot return the expected result for any reason, for example, the `malloc()` routine failed or none of the above-mentioned attempts to find an appropriate directory were successful, `NULL` will be returned.

ERRORS

Under the following conditions, the function `tempnam()` fails, and sets `errno` to:

`ENOMEM` if there is not enough space.

USAGE

Many applications prefer their temporary-files to have certain favorite initial letter sequences in their names. The *pfx* argument is used for this. This argument may be `NULL` or point to a string of up to five characters to be used as the first few characters of the temporary-filename.

The functions `tmpnam()` and `tempnam()` generate a different filename each time they are called.

Files created using these functions and either the `fopen()` routine [see fopen(BA_OS)] or the `creat()` routine [see creat(BA_OS)] are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to remove the file when its use is ended.

If called more than {TMP_MAX} times in a single process, these functions will start recycling previously used names.

Between the time a filename is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or `mktemp()` [see mktemp(BA_LIB)], and the filenames are chosen so as to render duplication by other means unlikely. The function `tmpnam()` uses `access()` [see access(BA_OS)] to determine whether the user is permitted to create a file in the named directory. This means that a setuid/setgid program trying to create a temporary file under a protected directory (one that the real UID/GID has no access to) will fail.

SEE ALSO

access(BA_OS), creat(BA_OS), fopen(BA_OS), malloc(BA_OS), mktemp(BA_LIB), tmpfile(BA_LIB), unlink(BA_OS).

LEVEL

Level 1.

NAME

trig: sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double  sin(double x);

double  cos(double x);

double  tan(double x);

double  asin(double x);

double  acos(double x);

double  atan(double x);

double  atan2(double y, double x);
```

DESCRIPTION

The functions `sin()`, `cos()`, and `tan()` return respectively the sine, cosine, and tangent of their argument, *x*, measured in radians.

The function `asin()` returns the arcsine in the range $-\pi/2$ to $\pi/2$ radians, of the argument *x*.

The function `acos()` returns the arccosine in the range `0` to π radians, of the argument *x*.

The function `atan()` returns the arctangent in the range $-\pi/2$ to $\pi/2$ radians, of the argument *x*.

The function `atan2()` returns the arctangent of y/x in the range $-\pi$ to π radians, using the signs of both arguments to determine the quadrant of the return value.

RETURN VALUE

If an input parameter is `NaN`, then the function will return `NaN` and set errno to `EDOM`.

When the absolute value of the argument to the functions `asin()` and `acos()` is greater than one and the value of the argument is not +-∞ or NaN, the return value will be an implementation-defined value (IEEE `NaN` or equivalent if available) and `errno` is set to `EDOM`.

When both arguments to the `atan2()` function are zero, the return value is implementation-defined and errno may be set to `EDOM`.

On a system that supports the IEEE 754 standard, if the value of x for cos(), sin(), tan(), asin(), or acos() is +-∞, these functions will return IEEE `NaN` and set errno to `EDOM`.

LEVEL

Level 1.

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

```
#include <search.h>

void *tsearch(const void *key, void **rootp,
     int(*compar)(const void *, const void *));

void *tfind(const void *key, void *const *rootp,
     int(*compar)(const void *, const void *));

void *tdelete(const void *key, void **rootp,
     int(*compar)(const void *, const void *));

void twalk(void *root, void(*action)(void **, VISIT, int));
```

DESCRIPTION

The functions `tsearch()`, `tfind()`, `tdelete()`, and `twalk()` manipulate binary search trees. All comparisons are done with a user-supplied function, *compar*. The comparison function is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to or greater than `0`, according to whether the first argument is to be considered less than, equal to or greater than the second argument, respectively. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The function `tsearch()` is used to build and access the tree. The value of *key* is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to **key* (the value pointed to by *key*), a pointer to this found datum is returned. Otherwise, **key* is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. The value of *rootp* points to a variable that points to the root of the tree. A `NULL` value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like `tsearch()`, `tfind()` will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, `tfind()` will return `NULL`. The arguments for `tfind()` are the same as for `tsearch()`.

The function `tdelete()` deletes a node from a binary search tree. The arguments are the same as for `tsearch()`. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree.

The function `twalk()` traverses a binary search tree. The value of *root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The value of *action* is the name of a user-defined routine to be invoked at each node. This routine is, in turn, called with three arguments.

The first argument is the address of the node being visited.

The second argument is a value from an enumeration data type, `VISIT` defined by the `<search.h>` header file. The values `preorder`, `postorder`, and `endorder`, indicate whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or the value `leaf` indicates that the node is a leaf.

The third argument is an integer that identifies the level of the node in the tree, with the root being level zero.

RETURN VALUE

`NULL` is returned by `tsearch()` if there is not enough space available to create a new node.

`NULL` is returned by `tsearch()`, `tfind()` and `tdelete()` if *rootp* is `NULL` on entry.

If the datum is found, both `tsearch()` and `tfind()` return a pointer to it. If not, `tfind()` returns `NULL`, and `tsearch()` returns a pointer to the inserted item. The function `tdelete()` returns a pointer to the parent of the deleted node, or `NULL` if the node is not found.

USAGE

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

The *root* argument to `twalk()` is one level of indirection less than the *rootp* arguments to `tsearch()` and `tdelete()`.

There are two nomenclatures used to refer to the order in which tree nodes are visited. The function `tsearch()` uses preorder, postorder and endorder to respectively refer to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

If the calling function alters the pointer to the root, results are unpredictable.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <stdio.oIf t0.5.10
```

```
        if (order == preorder || order == leaf) {
             printf("length=%d, string=%20s\n",
             (*(struct node **)node)->length,
             (*(struct node **)node)->string);
        }
    }

    main() {
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        int i = 0;

        while (gets(strptr) != NULL && i++ < 500) {
             nodeptr->string = strptr;
             nodeptr->length = strlen(strptr);
             (void) tsearch((void *)nodeptr,
                       &root, node_compare);
             strptr += nodeptr->length + 1;
             nodeptr++;
        }
        twalk(root, print_node);
    }
```

SEE ALSO

bsearch(BA_LIB), hsearch(BA_LIB), lsearch(BA_LIB).

LEVEL

Level 1.

NAME

ttyname, **isatty** – find name of a terminal

SYNOPSIS

```
#include <stdlib.h>

char *ttyname(int fildes);

int isatty(int fildes);
```

DESCRIPTION

ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

Return Values

ttyname returns a **NULL** pointer if *fildes* does not describe a terminal device in directory **/dev** or one of its subdirectories.

NOTICES

The value returned by **ttyname** points to static data whose content is overwritten by each call.

LEVEL

Level 1.

NAME

ungetc – push character back into input stdio-stream

SYNOPSIS

```
#include <stdio.h>

int ungetc(int c, FILE *strm);
```

DESCRIPTION

The function `ungetc()` inserts the character specified by *c* (converted to an `unsigned char`) into the buffer associated with an input stdio-stream. That character, *c*, will be returned by the next call to the `getc()` routine on that *strm*. The function `ungetc()` returns *c*, and leaves the file corresponding to *strm* unchanged. A successful call to `ungetc()` clears the end-of-file indicator for *strm*.

One character of pushback is guaranteed.

The value of the file position indicator for the stdio-stream after reading or discarding all pushed-back characters will be the same as it was before the characters were pushed back.

If the argument *c* equals `EOF`, the function `ungetc()` does nothing to the buffer and returns `EOF`.

The `fseek()`, `fsetpos()`, and `rewind()` routines [see fseek(BA_OS), fsetpos(BA_OS), and `rewind()` in fseek(BA_OS), respectively] erase all memory of inserted characters for the stdio-stream.

RETURN VALUE

Upon successful completion, the function `ungetc()` returns *c*; otherwise, it returns `EOF` if the character cannot be inserted.

SEE ALSO

fseek(BA_OS), fsetpos(BA_OS), getc(BA_LIB), setbuf(BA_LIB).

LEVEL

Level 1.

NAME

ungetwc – push **wchar_t** character back into input stream

SYNOPSIS

```
#include <stdio.h>
#include <widec.h>

wint_t ungetwc(wint_t c, FILE *stream);
```

DESCRIPTION

ungetwc inserts the wide (**wchar_t**) character *c* into the buffer associated with the input stream. That wide character, *c*, will be returned by the next **getwc** call on that stream. **ungetwc** returns *c*.

One wide character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered.

If *c* equals **(wchar_t)WEOF**, **ungetwc** does nothing to the buffer and returns **WEOF**.

fseek erases all memory of inserted characters.

Errors

ungetwc returns **WEOF** if it cannot insert the wide (**wchar_t**) character.

USAGE

Administrator.

SEE ALSO

fseek(BA_OS), **setbuf**(BA_LIB), **stdio**(BA_LIB), **getwc**(BA_LIB)

LEVEL

Level 1.

NAME

unlockpt – unlock a pseudo-terminal master/slave pair

SYNOPSIS

```
int unlockpt(int fildes);
```

DESCRIPTION

The function `unlockpt()` clears a lock flag associated with the slave pseudo-terminal device associated with its master pseudo-terminal counterpart so that the slave pseudo-terminal device can be opened. *fildes* is a file descriptor returned from a successful open of a master pseudo-terminal device.

RETURN VALUE

Upon successful completion, the function `unlockpt()` returns a value of `0`; otherwise, it returns a value of `-1`. A failure may occur if *fildes* is not an open file descriptor or is not associated with a master pseudo-terminal device.

SEE ALSO

grantpt(BA_LIB), open(BA_OS), ptsname(BA_LIB).

LEVEL

Level 1.

NAME

vfwprintf, **vwprintf**, **vswprintf** – print formatted wide character output of a variable argument list

SYNOPSIS

```
#include <stdarg.h>
#include <wchar.h>

int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);

int vwprintf(const wchar_t *format, va_list arg);

int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

DESCRIPTION

vfwprintf is equivalent to **fwprintf**, with the variable argument list replaced by an **arg** that has been initialized by the **va_start** macro.

vwprintf is equivalent to **wprintf**, with the variable argument list replaced by an **arg** that has been initialized by the **va_start** macro.

vswprintf is equivalent to **swprintf**, with the variable argument list replaced by an **arg** that has been initialized by the **va_start** macro. If copying takes place between objects that overlap, the behavior is undefined.

None of these functions invoke **va_end** or the passed **arg**.

Errors

vfwprintf and **vwprintf** return the number of wide characters transmitted or return a negative value if an error was encountered. **vswprintf** returns the number of wide characters written in the array, not counting the terminating null wide character, or returns a negative value if **n** or more wide character are requested to be generated.

USAGE

The following example shows the use of the **vfwprintf** function in a general error reporting routine:

```
#include <stdarg.h>
#include <wchar.h>

void error(wchar_t *function_name, wchar_t *format,...)
{
  va_list args;
  va_start(args, format);
  fwprintf(stderr, L"ERROR in %s: ", function_name);
  vfwprintf(stderr, format, args);
  va_end(args);
}
```

SEE ALSO

printf(BA_LIB), fwprintf(BA_LIB), putc(BA_LIB), scanf(BA_LIB), setlocale(BA_LIB), stdio(BA_LIB), write(BA_OS)

LEVEL

Level 1.

NAME

vfwscanf, **vwscanf**, **vswscanf** – convert formatted wide character input of a variable argument list

SYNOPSIS

```
#include <stdarg.h>
#include <wchar.h>

int vfwscanf(FILE *stream, const wchar_t *format, va_list arg);

int vwscanf(const wchar_t *format, va_list arg);

int vswscanf(wchar_t *s, const wchar_t *format,va_list arg);
```

DESCRIPTION

vfwscanf is equivalent to **fwscanf**, with the variable argument list replaced by an **arg** that has been initialized by the **va_start** macro.

vwscanf is equivalent to **wscanf**, with the variable argument list replaced by an **arg** that has been initialized by the **va_start** macro.

vswscanf is equivalent to **swscanf**, with the variable argument list replaced by an **arg** that has been initialized by the **va_start** macro.

None of these functions invoke **va_end** on the passed *arg*. If copying takes place between objects that overlap, the behavior is undefined.

Errors

vfwscanf, **vwscanf** and **vswscanf**1 return the number of wide characters transmitted or return a negative value if an error was encountered.

SEE ALSO

fwscanf(BA_LIB), putc(BA_LIB), scanf(BA_LIB), setlocale(BA_LIB), stdio(BA_LIB), write(BA_OS)

LEVEL

Level 1.

NAME

vprintf, vfprintf, vsprintf, vsnprintf – print formatted output of a variable argument list

SYNOPSIS

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

int vprintf(const char *format, va_list ap);

int vfprintf(FILE *stream, const char *format, va_list ap);

int vsprintf(char *s, const char *format, va_list ap);

int vsnprintf(char *s, size_t maxsize, const char *format, va_list ap);
```

DESCRIPTION

The functions `vprintf()`, `vfprintf()`, `vsprintf()`, and `vsnprintf()` are the same as `printf()`, `fprintf()`, `sprintf()`,and `snprintf()` respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the `<stdarg.h>` header file.

The `<stdarg.h>` header file defines the type `va_list` and a set of macros for advancing through a list of arguments whose number and types may vary. The argument *ap* to the vprint family of routines is of type `va_list`. This argument is used with the `<stdarg.h>` header file macros `va_start()`, `va_arg()` and `va_end()` [see `va_start()`, `va_arg()`, and `va_end()` in stdarg(BA_ENV)]. The **EXAMPLE** section below shows their use with `vprintf()`.

The macro `va_alist` is used as the parameter list in a function definition as in the function called `error()` in the example below. The macro `va_start(ap, parmN)`, where *ap* is of type `va_list`, and *parmN* is the rightmost parameter (just before `. . .`), must be called before any attempt to traverse and access unnamed arguments. Calls to `va_arg(ap, atype)` traverse the argument list. Each execution of `va_arg()` expands to an expression with the value and type of the next argument in the list *ap*, which is the same object initialized by `va_start`. The argument *atype* is the type that the returned argument is expected to be. The `va_end(ap)` macro must be invoked when all desired arguments have been accessed. (The argument list in *ap* can be traversed again if `va_start()` is called again after `va_end()`.) In the example below, `va_arg()` is executed first to return the function_name passed to `error()` and it is called again to retrieve the format passed to `error()`. The remaining `error()` arguments, *arg1*, *arg2*, ..., are given to `vfprintf()` in the argument *ap*.

RETURN VALUE

The functions `vprintf()`, and `vfprintf()` return the number of characters transmitted, or return `-1` if an error was encountered.

EXAMPLE

The following demonstrates how `vfprintf()` could be used to write an `error()` routine:

```
#include <stdio.h>
#include <stdarg.h>
/*
 *   error should be called like
 *         error(function_name, format, arg1, ...);
 */
void error(char *function_name, char *format, ...)
{
    va_list ap;

    va_start(ap, format);
    /* print out name of function causing error */
    (void) fprintf(stderr, "ERR in %s: ", function_name);
    va_arg(ap, char*);
    /* print out remainder of message */
    (void) vfprintf(stderr, format, ap);
    va_end(ap);
    (void) abort();
}
```

SEE ALSO

printf(BA_LIB), stdarg(BA_ENV).

LEVEL

Level 1.

NAME

vscanf, **vfscanf**, **vsscanf** – convert formatted input of a variable argument list

SYNOPSIS

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

int vscanf(const char *format, va_list ap);

int vfscanf(FILE *stream, const char *format, va_list ap);

int vsscanf(const char *s, const char *format, va_list ap);
```

DESCRIPTION

vscanf, **vfscanf** and **vsscanf** are the same as **scanf**, **fscanf**, and **sscanf** respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the **stdarg.h** header file.

The **stdarg.h** header file defines the type **va_list** and a set of macros for advancing through a list of arguments whose number and types may vary. [See **stdarg**(BA_ENV)].

Errors

These functions return the number of matched patterns, or return **EOF** if an error was encountered.

SEE ALSO

scanf(BA_LIB), **stdarg**(BA_ENV),

LEVEL

Level 1.

NAME

wconv: **towupper**, **towlower** – translate characters

SYNOPSIS

```
#include <wchar.h>

wint_t towupper(wint_t c);

wint_t towlower(wint_t c);
```

DESCRIPTION

If the argument to **towupper** is a wide character that is also a lowercase letter, the result is the corresponding uppercase letter. If the argument to **towlower** is a wide character that is also an uppercase letter, the result is the corresponding lowercase letter.

In the case of all other arguments, the return value is unchanged.

SEE ALSO

conv(BA_LIB), **wctype**(BA_LIB),

LEVEL

Level 1.

NAME

wcscat – concatenate two wide character strings

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

wcscat appends a copy of the wide character string *ws2*, including the **NULL** wide character, to the end of the wide character string *ws1*. The terminating null wide character at the end of *ws1* is overwritten by the initial wide character of *ws2*. The behavior is undefined if copying takes place between overlapping objects.

These functions do not check for an overflow condition of the array pointed to by *ws1*.

Return Value

wcscat returns *ws1*.

SEE ALSO

wcsncat(BA_LIB)

LEVEL

Level 1.

NAME

wcschr – scan a wide character string

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wint_t wc);
```

DESCRIPTION

wcschr scans the wide character string pointed to by *ws* for the wide character specified by *wc*. The null wide character terminating a string is considered to be part of the string.

Return Values

wcschr returns a pointer to the first occurrence of wide character *wc* in wide character string *ws*, or a null pointer if *wc* does not occur in the string.

SEE ALSO

wcsrchr(BA_LIB),

LEVEL

Level 1.

NAME

wcscmp – compare two wide character strings

SYNOPSIS

```
#include <wchar.h>

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

wcscmp makes a comparison between the wide character string pointed to by *ws1* and the wide character string pointed to by *ws2*.

Return Values

wcscmp compares its arguments and returns an integer less than, equal to, or greater than zero, depending on whether wide character string *ws1* is less than, equal to, or greater than wide character string *ws2*. The null wide character compares less than any other wide character.

SEE ALSO

wcsncmp(BA_LIB)

LEVEL

Level 1.

NAME

wcscoll – wide character string comparison using collating information

SYNOPSIS

```
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

wcscoll is part of the X/Open Portability Guide Issue 4 optional Enhanced Internationalization feature group. It compares the wide character string pointed to by **ws1** to the wide character string pointed to by **ws2**, which are both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

Return Values

wcscoll returns **0** and sets **errno** to **ENOSYS**.

Errors

In the following conditions, **wcscoll** fails and sets **errno** to:

EINVAL The **ws1** or **ws2** arguments contain wide character codes outside the domain of the collating sequence.

ENOSYS The function is not supported

USAGE

Since no return value is reserved to show an error, if you want to check for errors, you should set **errno** to **0**, call **wcscoll**, and then check **errno**. If it is non-zero, you can assume that an error has occurred.

Use **wcsxfrm** and **wcscmp** for sorting large lists of wide character strings.

SEE ALSO

strcoll(BA_LIB), **wcsxfrm**(BA_LIB)

LEVEL

Level 1.

NAME

wcscpy – copy a wide character string

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t *ws1,const wchar_t *ws2);
```

DESCRIPTION

wcscpy copies the wide string *ws2* to the array *ws1*, stopping after the null wide character has been copied. The behavior is undefined if copying occurs between overlapping objects.

Return Value

wcscpy returns *ws1*.

USAGE

Overlapping moves may cause unexpected results because the movement of wide character codes is implementation-dependent.

SEE ALSO

wchar(BA_ENV) **wcsncpy**(BA_LIB)

LEVEL

Level 1.

NAME

wcscspn – get length of complementary wide substring

SYNOPSIS

```
#include <wchar.h>

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

wcscspn determines the length of the maximum initial segment of the wide string pointed to by *ws1*. This string consists entirely of wide characters not included in the string pointed to by *ws2*.

Return Values

wcscspn returns the length of the segment.

SEE ALSO

wcsspn(BA_LIB)

LEVEL

Level 1.

NAME

wcsftime – convert date and time to wide character string

SYNOPSIS

```
#include <wchar.h>

int wcsftime(wchar_t *wcs, size_t size, const wchar_t *format, const
struct tm *timeptr);
```

DESCRIPTION

wcsftime puts wide character codes into the array pointed to by *wcs* as controlled by the string pointed to by *format*. It behavior is similar to **strftime**, except that the format and the result are wide character strings. Not more than **size** wide characters are placed into the array pointed to by **wcs**.

The behavior is undefined if copying takes place between objects that overlap.

Return Values

If the size of the resultant wide character codes inclusive of the terminating null wide character code is within the *size* limit, **wcsftime** returns the number of wide character codes in the array pointed to by *wcs*, exclusive of the terminating null wide character code. Otherwise, it returns zero and the contents of the array are indeterminate.

NOTICES

If the feature test macro **_XOPEN_SOURCE** is defined, then the following synopsis may be defined:

```
int wcsftime(wchar_t *wcs, size_t size, const char *format, const
struct tm *timeptr);
```

For conformance to XPG4's **wcsftmime**, an alternate interface is defined which we expect will be updated to match the above version in XPG's next release.

This version of **wcsftime** is allowed to set \rrno **to ENOSYS to show that the function is not implemented.**

SEE ALSO

strftime(BA_LIB), **wchar**(BA_ENV)

LEVEL

Level 1.

NAME

wcslen – obtain wide character string length

SYNOPSIS

```
#include <wchar.h>

size_t wcslen(const wchar_t *ws);
```

DESCRIPTION

wcslen returns the number of wide characters in wide character string *ws*, not including the terminating null wide character.

Return Values

wcslen returns the length of the string.

SEE ALSO

wchar(BA_ENV),

LEVEL

Level 1.

NAME

wcsncat – concatenate two wide character strings with bound

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

wcsncat appends at most *n* wide characters from the wide string *ws2* to the end of the wide string *ws1*. Wide characters that follow a null wide character are not copied. The null wide character at the end of *ws1* is overwritten by the initial wide character of *ws2*. A terminating null wide character is always appended to the result. The behavior is undefined if copying occurs between overlapping objects. This function does not check for an overflow condition of the array pointed to by *ws1*.

Return Values

wcsncat returns *ws1*.

SEE ALSO

wcscat(BA_LIB)

LEVEL

Level 1.

NAME

wcsncmp – compare two wide character strings with bound

SYNOPSIS

```
#include <wchar.h>

int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

wcsncmp compares not more than *n* wide characters from the array pointed to by *ws1* to the array pointed to by *ws2*. The function does not compare wide characters that follow a null wide character.

Return Values

wcsncmp compares its arguments and returns an integer less than, equal to, or greater than zero, depending on whether the wide string *ws1* is less than, equal to, or greater than the wide string *ws2*. The null wide character compares less than any other wide character.

SEE ALSO

wcscmp(BA_LIB),

LEVEL

Level 1.

NAME

wcsncpy – copy a wide character string with bound

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncpy(wchar_t *ws1,const wchar_t *ws2, size_t n);
```

DESCRIPTION

wcsncpy copies exactly *n* wide characters, truncating the wide string *ws2* or adding null wide characters to *ws1*, if necessary. Wide characters that follow a null wide character are not copied. The result will not be null-terminated if the length of *ws2* is *n* or more. If the array *ws2* points to is a wide character string that is shorter than *n* wide characters, the copy in the array pointed to by *ws1* is padded with null wide characters until a total of *n* wide characters is written. This function does not check for an overflow condition of the array pointed to by *ws1*.

Return Values

wcsncpy returns *ws1*.

USAGE

Overlapping moves may cause unexpected results because the movement of wide characters is implementation-dependent. If there is no null wide character in the first *n* wide characters of the array pointed to by *ws2*, the result will not be null-terminated.

SEE ALSO

wcscpy(BA_LIB)

LEVEL

Level 1.

NAME

wcspbrk – scan a wide character string for wide characters

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

wcspbrk returns a pointer to the first occurrence in the wide string *ws1* of any wide character from the wide string *ws2*, or a null pointer if there is no wide character from *ws2* in *ws1*.

Return Values

On completion, **wcspbrk** returns a pointer to the first wide character, or a null pointer if no wide character from *ws2* is found in *ws1*.

SEE ALSO

wcschr(BA_LIB), **wcsrchr**(BA_LIB),

LEVEL

Level 1.

NAME

wcsrchr – reverse wide character string scan

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsrchr(const wchar_t *ws, wint_t wc);
```

DESCRIPTION

wcsrchr scans the wide string *ws* for the last occurrence of the wide character *wc*. The null wide character terminating *ws* is considered to be part of the string.

Return Values

wcsrchr returns a pointer to the last occurrence of the wide character *wc* in the wide string *ws*, or a null pointer if *wc* does not occur in the string.

SEE ALSO

wcschr(BA_LIB),

LEVEL

Level 1.

NAME

wcsspn – obtain the length of a wide substring

SYNOPSIS

```
#include <wchar.h>

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

wcsspn returns the length of the initial segment of the wide string *ws1*, which consists entirely of the wide characters from the wide string *ws2*.

Return Values

wcsspn returns the length of the segment.

SEE ALSO

wcscspn(BA_LIB)

LEVEL

Level 1.

NAME

wcsstr, ‡wcswcs – find wide substring

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *ws1, const wchar_t *ws2);

wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

wcsstr locates the first occurrence in the wide character string pointed to by *ws1* of the sequence of wide characters (excluding the terminating null wide character) pointed to by *ws2*.

Return Values

Upon successful completion, **wcsstr** returns a pointer to the located wide character string, or a null pointer if the wide character string is not found.

wcsstr returns *ws1* if *ws2* points to a zero-length wide character string.

SEE ALSO

wcschr(BA_LIB),

LEVEL

Level 1.

wcswcs is designated Level 2 September 30, 1993.

wcswcs is only provided for XPG4 compatibility. It is anticipated that it will be removed in a future issue of XPG and of the SVID.

NAME

wcstod, **wcstof**, **wcstold** – convert wide string to floating point value

SYNOPSIS

```
#include <wchar.h>

double wcstod(const wchar_t *nptr, wchar_t **endptr);

float wcstof(const wchar_t *nptr, wchar_t **endptr);

long double wcstold(const wchar_t *nptr, wchar_t **endptr);
```

DESCRIPTION

wcstod returns, as a double-precision floating-point number, the wide character string pointed to by *nptr*. **wcstof** returns, as a single-precision floating-point number, the wide character string pointed to by *nptr*. **wcstold** returns, as a long double-precision floating-point number, the wide character string pointed to by *nptr*. Scanning occurs up to the first wide character that is unrecognized. The function recognizes an optional string that is composed of "white space" wide characters as defined by the **iswspace** function. The string is then followed by an optional sign then a sequence of digits optionally containing a decimal point character, followed by an exponential part (**e** or **E**) then another optional sign with an integer following it.

Also, instead of the regular decimal digit sequence, the string can be a **hexadecimal** floating value, an **infinity**, or a **NaN**. A hexadecimal floating value consists of **0x** or **OX** followed by a sequence of hexadecimal digits optionally containing a decimal point character, followed by a binary exponent part **p** or **P** then an optional sign with an integer following it. The exponent part must be present if no decimal point character is present. An infinity is specified by the string **inf** or **infinity** case insensitive. A **NaN** is specified by **nan** case insensitive, followed by an optional sequence of zero or more alphanumeric or underscore _ characters between a pair of parenthesis. If the value of *endptr* is not null, a pointer to the wide character terminating the scan is returned in the location pointed to by *endptr*.

Return Values

The function returns the value produced after the conversion process. If the function has not been performed then zero is returned and **errno** may be set to **EINVAL**.

If a correct value causes overflow, ±**HUGE_VAL** is returned, depending on the sign of the value, and **errno** is set to **ERANGE**.

If the value produced is correct but causes underflow, then zero will be returned with **errno** being set to **ERANGE**.

Errors

In the following conditions, these functions may fail and set **errno** to:

ERANGE The value produced after the conversion process would cause either an overflow or underflow.

EINVAL No conversion process could be carried out.

USAGE

Zero and ±**HUGE_VAL** can be returned as a correct value after the conversion process. However, they can also be returned on error. To check for an error condition, zero should be assigned to **errno** followed by a call to one of these functions

and then a check on **errno**. If the value of **errno** is non-zero it can be assumed that an error has occurred.

SEE ALSO

localeconv(BA_LIB), **scanf**(BA_LIB), **setlocale**(BA_OS), **wcstol**(BA_LIB)

LEVEL

Level 1.

NAME

wcstok – split a wide character string into tokens

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcstok(wchar_t *ws1,const wchar_t *ws2,wchar_t **savept);
```

DESCRIPTION

wcstok splits the wide string pointed to by *ws1* into tokens delimited by a wide character found in the wide string pointed to by *ws2*. *savept* points to a **wchar_t** pointer provided by the caller, in which **wcstok** stores information it needs to continue processing a particular wide string.

ws1 points to a wide string on the first call to **wcstok**, and is a null pointer on subsequent calls for the same wide string. When *ws1* is a null pointer, the value pointed to by *savept* is that set by the previous call to **wcstok** for the same wide string. Otherwise, the incoming value of the object pointed to by *savept* is ignored.

On the first call, **wcstok** searches for the first wide character which does not occur in the wide string pointed to by *ws2*. This wide character, if found, is the beginning of the first token. If no appropriate wide character is found, **wcstok** returns a null pointer, and there are no tokens in the wide string.

Starting at the first wide character of the token, **wcstok** searches for a wide character which does occur in the wide string pointed to by *ws2*. If an appropriate wide character is found, it becomes the end of the token, and is overwritten by a null wide character. The current token extends to the end of the wide string pointed to by *ws1* if no appropriate wide character is found. A null pointer is returned by any subsequent searches of the same wide string.

wcstok uses the pointer pointed to by *savept* to store enough information for subsequent calls to start searching just past the end of the token (if any) previously returned.

ws2 can point to a different wide character separator string for each call.

Return Values

On success, **wcstok** returns a pointer to the first wide character of a token. On failure, when no token is found, the function will return a null pointer.

NOTICES

The functionality of this interface is the same as that described previously, except an internal address is pointed to by *savept*, therefore no third argument is necessary.

SEE ALSO

wchar(BA_ENV)

LEVEL

Level 1.

NAME

wcstol – convert a wide character string to a long integer

SYNOPSIS

```
#include <wchar.h>

long wcstol(const wchar_t *nptr, wchar_t **endptr, int base);

long wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);
```

DESCRIPTION

wcstol returns, as a long integer, the value represented by the character string pointed to by *nptr*. **wcstoul** returns, as an unsigned long integer, the value represented by the character string pointed to by *nptr*. The string is scanned up to the first character inconsistent with the base. Leading ''white-space'' characters [as defined by **iswspace**] are ignored.

If the value of *endptr* is not a null pointer, a pointer to the wide character terminating the scan is returned in the location pointed to by *endptr*. If no integer can be formed, that location is set to *nptr*, and zero is returned.

If *base* is between 2 and 36, inclusive, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and a leading ''**0x**'' or ''**0X**'' is ignored if *base* is 16 and a leading **)b** or **0B** is ignored if **base** is **2**.

If *base* is zero, the string itself determines the base as follows: After an optional leading sign a leading zero indicates octal conversion, and a leading ''**0x**'' or ''**0X**'' hexadecimal conversion. Otherwise, decimal conversion is used.

Return Values

For **wcstol**, if the value represented by *nptr* would cause overflow, **LONG_MAX** or **LONG_MIN** is returned (according to the sign of the value), and **errno** is set to the value **ERANGE**.

For **wcstoul**, if the value represented by *nptr* would cause overflow, **ULONG_MAX** is returned, and **errno** is set to the value **ERANGE**.

If **wcstol** or **wcstoul** is given a *base* other than zero or 2 through 36, it returns zero and sets **errno** to **EINVAL**. Otherwise, **wcstol** and **wcstoul** return the represented value.

Errors

In the following conditions, **wcstol** fails and sets **errno** to:

EINVAL The value of *base* is not supported.

EINVAL No conversion could be performed.

ERANGE The value to be returned is not representable.

SEE ALSO

scanf(BA_LIB), **wcstod**(BA_LIB)

LEVEL

Level 1.

NAME

wcswidth – determine the number of column positions for a wide character string

SYNOPSIS

```
#include <wchar.h>

int wcswidth(const wchar_t *pwcs, size_t n);
```

DESCRIPTION

wcswidth determines the number of column printing positions needed for up to *n* wide characters in the wide string *pwcs*. Fewer than *n* wide characters will be processed only if a null wide character is encountered before *n* wide characters in *pwcs*.

Return Values

wcswidth returns either zero if *pwcs* is pointing to a null wide character code, or the number of column positions occupied by the wide character string pointed to by *pwcs*. **wcswidth** returns –1 if any wide character code in the wide character string pointed to by *pwcs* is not a printable wide character.

EXAMPLES

This example function, when passed a wide character string, calculates the number of column positions required and prints a diagnostic message.

```
#include <wchar.h>
#include <stdio.h>

. . .

int
print_width(const wchar_t *pwcs)
{
     int width;
     size_t len;

     len = wcslen(pwcs);
     if (len > 0)  {
          width = wcswidth (pwcs, len);
          if (width == -1)
             (void) printf("non printable character\n");
          else
             (void) printf("Wide string width=%d\n",width);
          return (1);
     }
     (void) printf("zero length wide character string\n");
     return (0);

}
```

SEE ALSO

wchar(BA_DEV), **wcwidth**(BA_LIB)

LEVEL

Level 1.

NAME

wcsxfrm – wide character string transformation

SYNOPSIS

```
#include <wchar.h.h>

size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

wcsxfrm is part of the X/Open Portability Guide Issue 4 optional Enhanced Internationalization feature group.

wcsxfrm transforms the wide character string pointed to by **ws2** and places the resulting wide character string into the array pointed to by **ws1**. The transformation does the following:

If **wcscmp** is applied to two transformed wide strings, it returns a value greater than, or equal to, zero, corresponding to the result of **wcscoll** applied to the same two original wide character strings.

No more than *n* wide-character codes are placed into the resulting array pointed to by **wcs1**, including the terminating null wide-character code. If *n* is zero, **wcs1** can be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Return Values

wcsxfrm returns the length necessary to hold the entire transformed wide character string, not including the terminating null wide-character code. If the value returned is *n* or more, the contents of the array pointed to by *ws1* are indeterminate. **wcsxfrm** returns **-1** and sets **errno** to **ENOSYS**.

Errors

wcsxfrm may fail if

EINVAL The **ws1** or **ws2** arguments contain wide character codes outside the domain of the collating sequence.

ENOSYS The function is not supported

USAGE

Since no return value is reserved to show an error, if you want to check for errors, you should set **errno** to **0**, call **wcscoll**, and then check **errno**. If it is non-zero, you can assume that an error has occurred.

SEE ALSO

strxfrm(BA_LIB), **wchar**(BA_DEV), **wcscoll**(BA_LIB)

LEVEL

Level 1.

NAME

wctob – wide character to byte conversion

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t c);
```

DESCRIPTION

wctob determines whether *c* corresponds to a member of the extended character set whose multibyte character representation is as a single byte when in the initial shift state.

Return Values

wctob returns **EOF** if *c* does not correspond to a multibyte character with length one; otherwise, it returns the single byte representation.

LEVEL

Level 1.

NAME

wctype: **iswalpha**, **iswupper**, **iswlower**, **iswdigit**, **iswxdigit**, **iswalnum**, **iswspace**, **iswpunct**, **iswprint**, **iswgraph**, **iswcntrl** – test wide characters for a specified class

SYNOPSIS

```
#include <wchar.h>

int iswalpha(wint_t wc);

int iswupper(wint_t wc);

int iswlower(wint_t wc);

int iswdigit(wint_t wc);

int iswxdigit(wint_t wc);

int iswalnum(wint_t wc);

int iswspace(wint_t wc);

int iswpunct(wint_t wc);

int iswprint(wint_t wc);

int iswgraph(wint_t wc);

int iswcntrl(wint_t wc);
```

DESCRIPTION

iswalpha(*wc***)**	*wc* is an alphabetic wide character.
iswupper(*wc***)**	*wc* is an uppercase wide character.
iswlower(*wc***)**	*wc* is a lowercase wide character.
iswdigit(*wc***)**	*wc* is a wide character representing a digit.
iswxdigit(*wc***)**	*wc* is a wide character representing a hexadecimal digit.
iswalnum(*wc***)**	*wc* is an alphanumeric wide character.
iswspace(*wc***)**	*wc* is a wide character representing a white space character.
iswpunct(*wc***)**	*wc* is a wide character representing a punctuation character.
iswprint(*wc***)**	*wc* is a wide character representing a printing character including space.
iswgraph(*wc***)**	*wc* is a wide character like above but does not include white space.
iswcntrl*wc*	*wc* is a control characters (not printable)

Return Values

If *wc* matches the classification of the called function, nonzero is returned. Otherwise, zero is returned. locale (category **LC_CTYPE**).

LEVEL

Level 1.

NAME

wcwidth – determine the number of column positions for a wide character

SYNOPSIS

```
#include <wchar.h>

int wcwidth(wint_t wc);
```

DESCRIPTION

wcwidth determines the number of column printing positions that are needed by the wide character *wc*.

Return Values

wcwidth returns zero if *wc* is a null wide character, or the number of column printing positions the wide character *wc* occupies. **wcwidth** returns -1 if *wc* does not correspond to a valid, printable wide character.

EXAMPLE

Here is a program that reads a wide character from standard input and prints the width of the character.

```
#include <wchar.h>
#include <stdio.h>

main()
{
     int x;
     wint_t wc;

     if ((wc=fgetwc(stdin)) != WEOF) {
          x=wcwidth(wc);
          if (x==-1)
            (void) printf("Character not printable\n");
          else
            (void) printf("Character width=%d\n",x);
          exit(0);
     }
     (void) printf("Error encountered reading character\n");
     exit(2);
}
```

SEE ALSO

wchar(BA_DEV), **wcswidth**(BA_LIB)

LEVEL

Level 1.

NAME

wordexp, wordfree – perform word expansions

SYNOPSIS

```
#include <wordexp.h>

int wordexp(const char *string, wordexp_t *pword,int flags);

void wordfree(wordexp_t *pword);
```

DESCRIPTION

These functions are part of the X/Open Portability Guide Issue 4 optional POSIX2 C-Language Binding feature group.

wordexp performs word expansions and places the list of expanded words into the structure pointed to by **pword**.

Return Values

wordexp returns **WRDE_NOSYS** and sets **errno** to **ENOSYS**.

wordfree returns and sets **errno** to **ENOSYS**.

Errors

In the following conditions, **wordexp** returns and sets **errno** to:

WRDE_BADCHAR One of the unquoted characters appears in *words* in an inappropriate context.

WRDE_BADVAL Reference to an undefined shell variable when **WRDE_UNDEF** is set in *flags*.

WRDE_CMDSUB Command substitution requested when **WRDE_NOCMD** was set in *flags*.

WRDE_NOSPACE Attempt to allocate memory failed.

WRDE_SYNTAX Shell syntax error.

USAGE

wordexp should be used by an application that wants to do all the shell's expansions on a word or words obtained from a user. If the application prompts for a file name and then uses **wordexp** to process the input, you could respond with anything that would be valid input to the shell.

The **WRDE_NOCMD** flag prevents you from executing shell commands. Not allowing unquoted shell special characters also prevents unwanted side effects such as executing a command or writing a file.

SEE ALSO

fnmatch(BA_LIB), **glob**(BA_LIB),

LEVEL

Level 1.

Base System Devices Introduction

This section contains an overview of the STREAMS I/O Interfaces, followed by the BA_DEV manual pages.

STREAMS I/O Interfaces Overview

STREAMS is a general, flexible facility for development of communication services. It supports development ranging from complete networking protocol suites to individual device drivers by defining standard interfaces for character input/output within the kernel. The standard interfaces and associated tools enable modular, portable development and easy integration of high performance network services and their components. STREAMS provides a broad framework that does not impose any specific network architecture. It implements a user interface consistent and compatible with the character I/O mechanism.

The power of STREAMS resides in its modularity. The design reflects the layering characteristics of contemporary networking architectures such as Open Systems Interconnection (OSI), Systems Network Architecture (SNA), Transmission Control Protocol/Internet Protocol (TCP/IP), and XEROX Network Systems (XNS) (XEROX is a registered trademark of Xerox Corporation). For these protocol suites, developers have traditionally faced problems arising from lack of relevant standard interfaces. STREAMS defines standard mechanisms for implementing protocols in "modules". Each module represents a set of processing functions and communicates with other modules via a standard interface. From user-level, kernel-resident modules can be dynamically selected and interconnected to implement any rational processing sequence. Modularity allows these advantages:

- User-level programs can be independent of underlying protocols and physical communication media.
- Network architectures and higher-level protocols can be independent of underlying protocols, drivers, and physical communication media. This enables customers to retain their investment in application software as they migrate to different networking environments.
- Higher level services can be created by selecting and connecting lower level services and protocols.
- Protocol module portability is enhanced by well defined structure and interface standards.

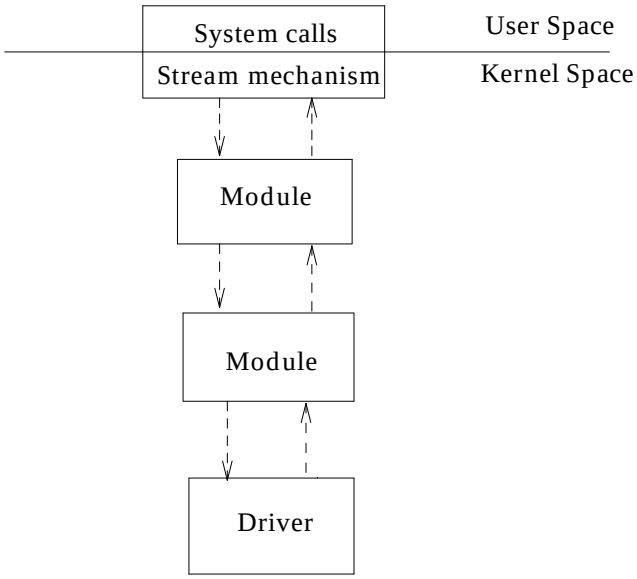
- Terminal subsystems can have customized line discipline modules.

Implementing networking facilities and communication components under STREAMS allows efficient, open-ended products.

STREAMs Fundamentals

"STREAMS" refers to the mechanism consisting of operating system service routines, kernel resources, and kernel utility routines. A stream, as illustrated in figure 1, is a full duplex processing and data transfer path in the kernel that is created through an application of the STREAMS mechanism.

Figure 7-1: Basic Stream



A stream implements a connection between a driver in kernel space and a process in user space. It provides a general character input/output (I/O) interface for user processes. STREAMS I/O is based on messages. Messages flow in both directions in a stream. Each module represents processing functions to be performed on the contents of messages flowing into the module on the stream. Each module is self-contained and functionally isolated from any other component in the stream except its two neighboring components. A module communicates with

its neighbors by passing messages. The module receives the message, inspects the type, and processes it or just passes it on. A module can function, for example as, a communication protocol, line discipline, or data filter.

There are many message types used by STREAMS modules. They can be classified according to queueing priority. Every message has a priority band associated with it. Messages may be normal, priority, or high-priority. Normal messages have a priority band of zero and are always placed at the end of the queue following all other messages in the queue. High-priority messages are always placed at the head of a queue but after any other high-priority messages already in the queue. By convention they are not affected by flow control and their priority band is ignored. They are high-priority by virtue of the message type. Priority messages are always placed on the queue as indicated by their priority band. They are placed after any messages in the same priority band already on the queue. High-priority and priority messages are used to send control and data information outside the normal flow control constraints. Priority messages enable support of "expedited" or "urgent" data which are needed for various networking protocols. Each priority band is subject to separate flow control from other priority bands. To prevent congestion and resource waste due to lack of flow control with high-priority messages, only one high-priority message may be placed in the stream head read queue at a time.

A user may access STREAMS messages that contain a data portion, control portion, or both. The data portion is that information which is sent out over the network and the control information is used by the local STREAMS modules. The other types of messages are used between modules and not accessible to users. Messages containing only a data portion are accessible via `putmsg()`, `putpmsg()`, `getmsg()`, `getpmsg()`, `read()`, and `write()` routines. Messages containing a control portion with or without a data portion are accessible via calls to `putmsg()`, `putpmsg()`, `getmsg()`, and `getpmsg()`.

The interface between a user process and STREAMS is compatible with the pre-STREAMS character I/O facilities.

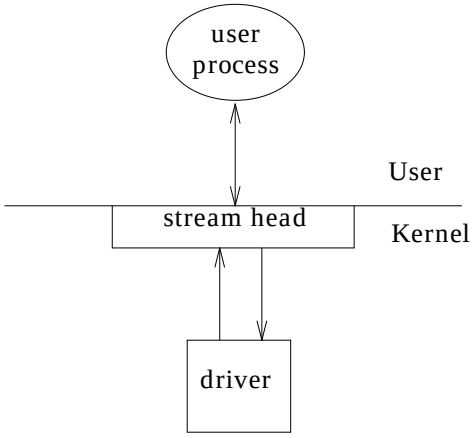
Accessing Streams

User access to STREAMS is provided through a set of operating system service routines. These include the traditional `open()`, `close()`, `read()`, `write()`, and `ioctl()` operating system service routines as well as the `putmsg()`, `putpmsg()`, `getmsg()`, `getpmsg()`, and `poll()` routines.

Setting Up a Stream

Like conventional drivers, the STREAMS-based driver occupies a node in the file system and may be "opened" and "closed". When a STREAMS-based device is opened, a stream is automatically set up. As shown in Figure 2, this open sets up a stream with an internal module called the "stream head" closest to the user and the device driver downstream from the stream head.

Figure 7-2: Setting Up a Stream

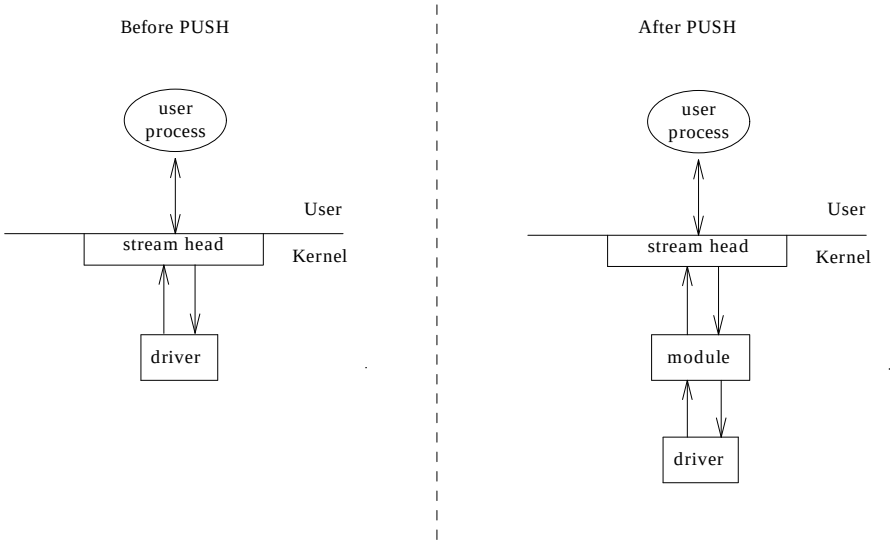


The stream then consists of the stream head and a driver. To add other modules to the stream, the user calls the `ioctl()` operating system service routine to "PUSH" a module.
The syntax for this `ioctl()` command is

```
ioctl(fd, I_PUSH, "name")
```

where *fd* is the file descriptor of the open stream, `I_PUSH` is the command, and "*name*" is the name of the module to be pushed. The number of modules that may be pushed onto a stream is a configurable quantity. A new module is always pushed just below the stream head so the order of "pushes" is important. After the module is pushed, the stream looks as shown in the figure below:

Figure 7-3: Before and After a Module is Pushed



The user may "POP" modules off a stream using the `ioctl()` command

```
ioctl(fd, I_POP, 0)
```

This routine removes the module most recently added to the stream designated by the file descriptor *fd*; this is always the intermediate module closest to the stream head. At the user-level, drivers are operationally distinct from other modules; drivers are explicitly opened by device pathname, while modules are "pushed" onto the stream by module name. Device pathnames are ordinary system filenames, but pushable modules' names are internal to the system and are not visible in the file system.

Sending and Receiving STREAMS Messages

In order to send and receive STREAMS messages that contain control information, the routines `getmsg()`, `getpmsg()`, `putmsg()`, and `putpmsg()` must be used. These differ from `read()` and `write()` in that the traditional routines can access STREAMS messages containing only data, while `getmsg()`, `getpmsg()`, `putmsg()`, and `putpmsg()` can access messages containing a control portion, data portion, or both. The control portion is used to carry interface information between modules and drivers.

As an example, the transport functions of the OPEN SYSTEMS NETWORKING INTERFACES use `putmsg()` to send service requests (*e.g.*, to establish a connection), with or without data, to the underlying STREAMS-based transport protocol. `getmsg()` is used by the transport functions to receive information back.

Polling STREAMS

The `poll()` routine provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open files; this section will describe how `poll()` can be used in conjunction with files that are streams. `poll()` identifies those streams on which a user can send or receive messages or on which certain events have occurred. The syntax for `poll()` is as follows:

```
int poll(pollfds, nfds, timeout)
```

where *nfds* specifies the number of file descriptors to be examined, *timeout* specifies the number of msec that `poll()` should wait for an event to occur, and *pollfds* is an array of `pollfd` structures where each structure contains the following members:

```
int fd;         /* file descriptor */
short events;   /* requested events */
short revents;  /* returned events */
```

These structures specify the file descriptors to be examined and the events of interest for each file descriptor. `fd` specifies an open file descriptor and `events` and `revents` are bitmasks constructed by OR-ing any combination of the events specific to the `poll()` operating system service routine.

For each element of the array pointed to by *pollfds*, `poll()` examines the given file descriptor for the event(s) specified in `events`.

The results of the `poll()` query are stored in the `revents` field in the `pollfd` structure. Bits are set in the `revents` bitmask to indicate which of the requested events are true. If none are true, none of the specified bits is set in `revents` when the `poll()` call returns.

If none of the defined events have occurred on any selected file descriptor, `poll()` waits at least *timeout* msec for an event to occur on any of the selected file descriptors. If the value of *timeout* is `0`, `poll()` returns immediately, effectively polling the file descriptors. If the value of *timeout* is `-1`, `poll()` blocks until a requested event occurs or until the call is interrupted.

Multiplexing in STREAMS

Until now, STREAMS has been described as linear connections of modules, where each invocation of a module is connected to at most a single upstream module and a single downstream module. While this configuration is suitable for many applications, others require the ability to multiplex STREAMS in a variety of configurations. Typical examples are internetworking protocols, which might route data over several subnetworks, or terminal window facilities.

STREAMS provides the capability to dynamically build, maintain, and dismantle multiplexing configurations. Two types of multiplexing are supported by STREAMS. The first type allows user processes to connect multiple streams to a single driver from *above*. This configuration can be established by opening multiple minor devices of the same driver, and does not require any special STREAMS facilities. The second multiplexing type allows user processes to connect multiple streams *below* a pseudo-driver. This configuration must contain a multiplexing pseudo-driver recognized by STREAMS as having special characteristics. A special set of `ioctl()` commands is used to establish this multiplexing configuration. STREAMS allows a user to build complex, multi-level configurations by cascading multiplexing streams below one another.

Setting Up a Multiplexer

A multiplexing driver is a pseudo-device, and is treated like any other software driver. It has a node in the file system, and is opened just like any other STREAMS device driver. The `open()` call establishes a single stream "above" the multiplexer, and the process that opened the multiplexer is

returned a file descriptor that can be used to access the stream that was opened. The file descriptor `fd0` in Figure 4 is an example of this.

Next, one of the drivers that is to exist "below" the multiplexer is opened. Once again, this is a driver, and is opened like any other system device. The `open()` operating system service routine is used to open the driver, a stream is established between the driver and a stream head, and the process that issued the `open()` call is returned a file descriptor that can be used to access the stream connected to the driver (*e.g.*, `fd1` in Figure 4).

If the eventual multiplexing configuration is to have intermediate protocol or line-discipline modules in the stream between the driver just opened and the multiplexer (*e.g.*, between the MUX driver and Driver1 in the "After" section of Figure 4), these modules should be added at this time to the stream just opened, using the `I_PUSH ioctl()` command. The "push" operation must be done before the driver is attached below the multiplexer because, once connected, `ioctl()` commands cannot be issued to the bottom driver in the normal way.

The driver that was just opened is then connected below the multiplexing driver that was opened first. This is done using the `I_LINK` command of the `ioctl()` operating system service routine; the complete sequence is given here:

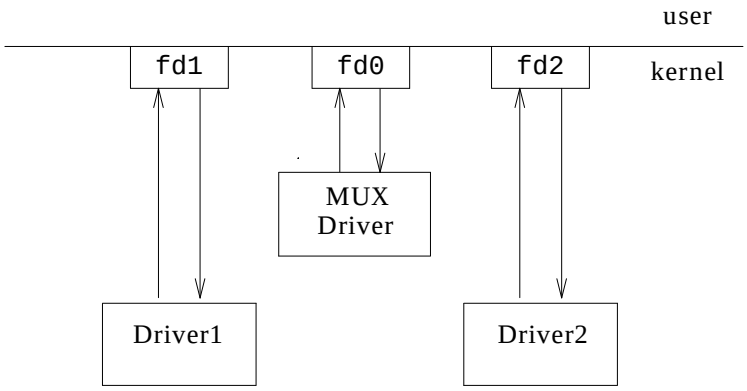
```
fd0 = open("/dev/MUXdriver", oflag);
fd1 = open("/dev/Driver1", oflag);
mux_id = ioctl(fd0, I_LINK, fd1);
```

Here, the variable `fd0` is the file descriptor for the stream connected to the multiplexing driver, and `fd1` is the file descriptor for the stream connected to another driver. It should be noted that in the `ioctl()` call the placement of the first argument (`fd0`) and the third argument (`fd1`) is important; the first argument *must* be the file descriptor of the stream connected to the multiplexing driver. (See Figure 4.) The value `mux_id` is returned by the operating system service routine; it is used by the multiplexing module to identify the stream just connected.

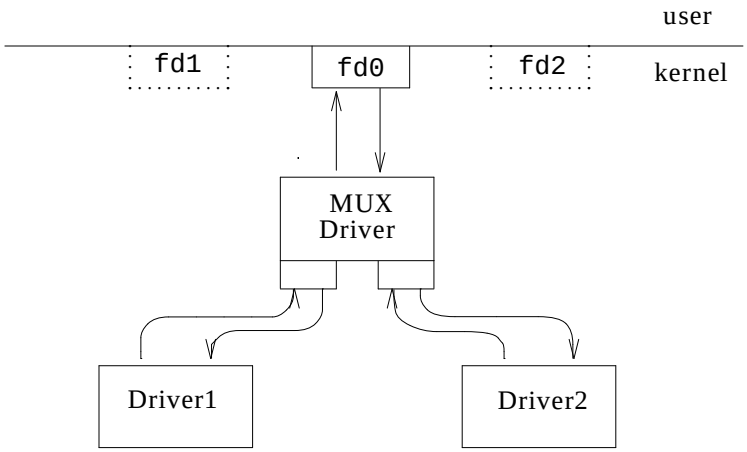
Figure 4 shows two drivers and a multiplexing driver before and after the two drivers have been linked below the multiplexer.

Figure 7-4: A Multiplexing Configuration Before and After 2 `I_LINK` `ioctl()` Calls

BEFORE:



AFTER:



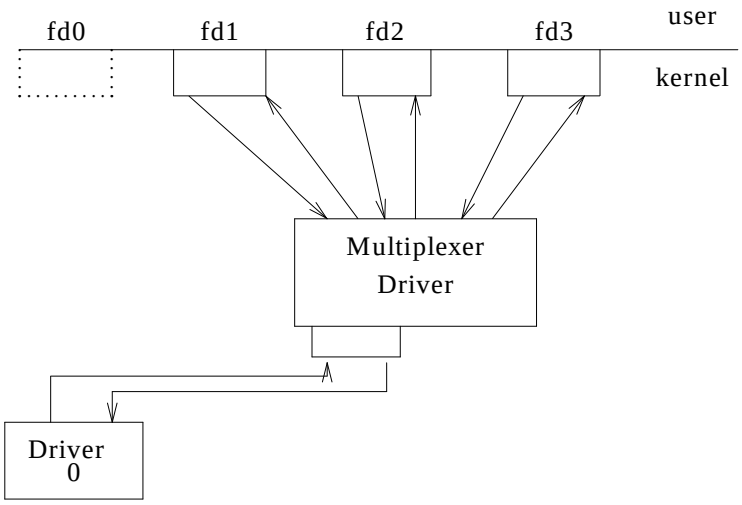
Other device drivers are opened and linked below the multiplexing driver in the same way, as in the example shown in Figure 4:

```
/* open another driver */
fd2 = open("/dev/driver2", oflag);
/* link it below the MUX */
mux_id2 = ioctl(fd0, I_LINK, fd2);
```

The number of streams that can be "linked" to a multiplexer depends on the particular multiplexer, and it is the responsibility of the multiplexer to keep track of the streams linked to it. However, only one `I_LINK` operation is allowed for each "lower" stream; a single stream cannot be linked below two multiplexers simultaneously.

The order in which the streams in the multiplexing configuration are opened is unimportant. It is only necessary that the two streams referenced as arguments to the `I_LINK ioctl()` are both open when the `I_LINK ioctl()` command is issued. Once the configuration is established, the file descriptors that point to the "bottom" device drivers (*e.g.*, `fd1` and `fd2` in Figure 4) can be closed without affecting the way the multiplexer works; these closes will not cause the drivers to be unlinked from the multiplexer. If these file descriptors (`fd1` and `fd2` in Figure 4) are not closed, the multiplexer will work as expected, but all subsequent `read()`, `write()`, `ioctl()`, `poll()`, `putmsg()`, `putpmsg()`, `getmsg()`, and `getpmsg()` OS service routine calls issued to `fd1` and `fd2` will fail.

Figure 7-5: Three STREAMS Converging on One Device Driver



Building a multiplexer that connects several streams to a single driver, as in Figure 5, is similar, except that only one driver is linked below the multiplexer. Additional streams above the multiplexer would be established by issuing repeated `open()` calls to the multiplexer on "related" minor devices. Again, the way the multiplexer handles these repeated calls to `open()` is multiplexer-dependent, as is the number of streams that a particular multiplexer will successfully handle.

More complex multiplexing configurations can also be created. It is possible to combine the examples of Figures 4 and 5 to create a configuration with many streams above and many drivers linked below the multiplexer. STREAMS imposes no restrictions on the number of multiplexing drivers that may be included in a multiplexing configuration or on the number of multiplexers that data can pass through when moving from one end of the multiplexing configuration to the other.

Another type of link, called a "persistent link", can also be created in a multiplexing configuration. Two new `ioctl()` commands, `I_PLINK` and `I_PUNLINK`, are used to create and remove such "persistent" links. The syntax for these commands is the same as for `I_LINK` and `I_UNLINK`. However, these persistent links are not associated with the stream above the multiplexer. A `close()` or `I_UNLINK` would not disconnect the persistent links. In Figure 4, if the link to Driver1 is a persistent link, the file descriptor, `fd0`, associated with the stream above the MUX Driver, can be closed without dismantling the persistent link below. Other users can come in and open MUXdriver and send data to Driver1 since the persistent link to Driver1 remains intact.

In a multi-level multiplexing configuration where persistent links exist below a multiplexer whose stream is connected to the above multiplexer by regular links, closing the file descriptor associated with the controlling stream will remove the regular link but not the persistent links below it. Regular links are also allowed to exist below a multiplexer whose upper stream is connected via a persistent link. In this case, the regular links would be removed if the persistent link above them is removed, and if there were no open references to the lower streams.

Dismantling a Multiplexer

Multiplexing configurations are taken apart using the `ioctl() I_UNLINK` or `I_PUNLINK` command. Each of the bottom drivers linked below the multiplexing driver (*e.g.*, Driver1 and Driver2 in Figure 4) can be individually disconnected:

```
ioctl(fd0, I_UNLINK, mux_id);
```

Here, `fd0` is the file descriptor pointing to a stream connected to the multiplexing driver, and `mux_id` is the identifier that was returned by the `ioctl() I_LINK` command when one of the bottom drivers was linked to the multiplexing driver. Each bottom driver can be disconnected individually in this way, or a special

mux_id value of MUXID_ALL will disconnect all bottom modules from the multiplexer simultaneously. This unlinking occurs automatically on the last close of the top stream through which the lower streams were linked under the multiplexer; all these bottom streams are then unlinked.

To disconnect a persistent link, one would have to first open the driver to obtain a file descriptor fd0, if it had been closed, and then call ioctl() with I_PUNLINK as the command using the mux_id that had been returned on the previous I_PLINK command. This call removes the persistent link in between the multiplexer referenced by fd0 and the stream to the driver designated by mux_id. A call with a mux_id value of MUXID_ALL will unlink all persistent links below the multiplexing driver referenced by fd0.

The use of I_PLINK and I_PUNLINK should not be intermixed with that of I_LINK and I_UNLINK. Any attempt to unlink a regular link via I_PUNLINK or to unlink a persistent link via I_UNLINK will fail.

Multiplexed Data Routing

Processes use the normal read(), write(), getmsg(), getpmsg(), putmsg(), and putpmsg() operating system service routines to read data from and write data to an upper stream connected to the multiplexer. When these data are routed through a multiplexer, the multiplexer must use its own criteria to route the data moving in both directions. For example, a protocol multiplexer might use protocol address information found in a protocol header to determine over which subnetwork a given packet should be routed. It is the multiplexing driver's responsibility to define its routing criteria.

One option available to the multiplexer is to use the "mux id" value to determine which stream to route data to. The multiplexer has access to this value, and the I_LINK ioctl() command returns this value to the user. The multiplexer can therefore specify that the "mux id" value accompany the data routed through it.

Pipe Fundamentals

A pipe is a mechanism that provides a communication path between multiple processes. It implements a user interface consistent and compatible with the character I/O mechanism.

A STREAMS-based pipe, as shown in Figure 6, is a full duplex processing and data transfer path in the kernel that is created by a user process invoking the pipe() routine. A pipe implements a connection between the kernel and one or more user processes. A STREAMS-based pipe supports capabilities beyond those of the traditional pipe but has maintained the semantics of the traditional pipe. Because of the STREAMS framework, a user can push processing modules, poll(), and pass file descriptors across these pipe connections.

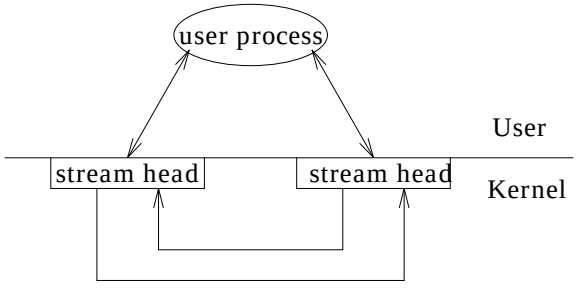
The remainder of this section will use the term pipe to refer to a STREAMS-based pipe.

Creating and Accessing Pipes

A user process creates a pipe via the `pipe()` operating system service routine which returns two file descriptors, `fd[0]` and `fd[1]`, that are opened for both reading and writing. Data written to `fd[0]` can be read from `fd[1]` and data written to `fd[1]` can be read from `fd[0]`. Unlike STREAMS-based drivers or pseudo drivers, a pipe is not an object in the file system name space. A user process accesses the pipe through one of these two file descriptors that represent each end of the pipe.

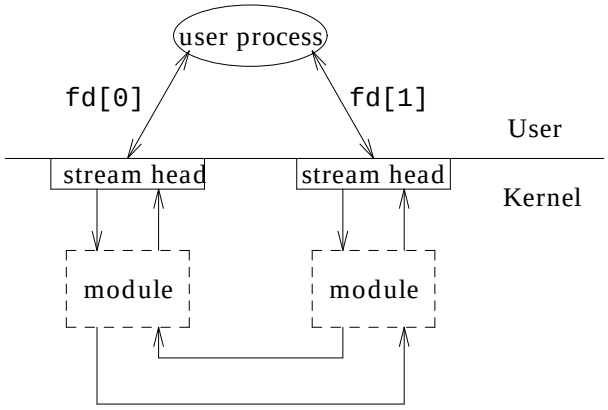
When a pipe is created via the `pipe()` routine, two streams are automatically set up, each only consisting of an internal stream head module. As shown in Figure 6, a pipe represents two separate streams, with both streams attached in such a way that messages flow in either direction, from one stream head to the other.

Figure 7-6: Basic STREAMS-based Pipe



Other modules can be added to the pipe if the user invokes `ioctl()` to "PUSH" the modules, as shown in Figure 7.

Figure 7-7: Pushing Modules on a Pipe



Named Streams

Some applications may find it helpful to be able to dynamically associate a stream or STREAMS-based pipe with an existing object in the file system name space. For example, a server process may create a pipe, name one end of the pipe, and permit unrelated processes to communicate with it over the named pipe.

A STREAMS file descriptor can be named by attaching that file descriptor to an object in the file system name space. The routine used to name a STREAMS file descriptor is `fattach()` which has the following interface:

`fattach (int` *fildes*, `char *`*path*)

fildes must be an open file descriptor that refers to either a STREAMS-based pipe or a STREAMS device driver (or pseudo device driver). This discussion describes the scenario where *fildes* represents a STREAMS-based pipe. *path* is an existing object in the file system name space (e.g. regular file, directory, character special file, etc.) and cannot already have a STREAMS file attached to it. In addition, *path* must not be the mount point for a file system nor the root of a file system. To attach the file descriptor, the user must be the owner of *path* with write permission or must be a process with the appropriate privileges.

If *path* is currently in use at the time `fattach()` is executed, those user processes accessing *path* will not be interrupted and any data that was associated with *path* before the call to `fattach()` will continue to be accessible by those processes.

After a file descriptor is named, all subsequent operations (*e.g.* `open()`) on *path* will operate on the named stream. Thus, it is possible that a user process can have one file descriptor pointing to the data associated with *path* and another file descriptor pointing to the named STREAMS-pipe.

Once the stream is named, `stat()` on *path* will project the information for the STREAMS-file. That is, if the named stream is a pipe, the `stat()` information will show that *path* is a pipe. If the STREAMS file is a device driver (or pseudo device driver) *path* will show the information for the devices. The attributes of the named stream[see stat(BA_OS)] are initialized as follows: the permissions, user ID, group ID, and times are set to those of *path*, the number of links is set to 1, and the size and device indentifier are set to those of the streams device associated with *fildes*. Once the stream is named, the user can issue `chmod()`, `chown()` to alter the attributes of the named stream and not

affect the original attributes of *path* nor the original attributes of the STREAMS-file.

The size represented in the `stat()` information will reflect the number of unread bytes of data currently at the stream head. This size is not necessarily the number of bytes written to the STREAM.

A STREAMS-based file descriptor can be attached to many different paths at the same time, *i.e.* a stream can have several names attached to it. The modes, ownership and permissions of these paths may vary. However, operations on any of these paths will access the same stream.

Since named streams are STREAMS devices, processes can push modules, `poll()`, pass file descriptors, or do any other STREAMS operations on them.

To disassociate a filename from a named stream, the `fdetach()` routine is invoked with the following interface:

```
fdetach (char *path)
```

where *path* is the name of the previously attached object. The user must be the owner of *path* or a user with the appropriate privileges. If processes have the named stream open at the time of the call to `fdetach()`, these processes are not affected and continue to access the named stream.

The original permission, mode and ownership are restored to the state prior to naming. In addition, the type and the size of the object reflect the object itself, as it appears in the file system. Subsequent operations on *path* will access the file system object and no longer access the named stream. If only one end of the pipe is attached, the last close of the other end (for example the process closes down) will cause the attached end to be automatically detached. If, however, the named stream is a device and not a pipe, the last close of the file will not cause the stream to be detached. A process has to invoke `fdetach()` to detach the stream.

Passing File Descriptors

Named stream pipes are especially useful for passing file descriptors between unrelated processes. A user process can send a file descriptor to another process by invoking `ioctl()` on one end of a named stream pipe with the `I_SENDFD` command. This sends a message containing a file pointer to the stream head at the other end of the pipe. Another process can retrieve that message containing the file pointer by invoking `ioctl()` on the other end of the pipe with the `I_RECVFD` command.

Base System Devices

This following section contains the manual pages for the BA_DEV routines.

FINAL COPY
June 15, 1995
File: ba_dev.cov
svid

NAME

connld – line discipline for unique stream connections

DESCRIPTION

connld is a STREAMS-based module that provides unique connections between server and client processes. It can only be pushed [see **streamio**(BA_DEV)] onto one end of a STREAMS-based pipe that may subsequently be attached to a name in the file system name space. After the pipe end is attached, a new pipe is created internally when an originating process attempts to **open**(BA_OS) or **creat**(BA_OS) the file system name. A file descriptor for one end of the new pipe is packaged into a message identical to that for the **ioctl I_SENDFD** [see **streamio**(BA_DEV)] and is transmitted along the stream to the server process on the other end. The originating process is blocked until the server responds.

The server responds to the **I_SENDFD** request by accepting the file descriptor through the **I_RECVFD ioctl** message. When this happens, the file descriptor associated with the other end of the new pipe is transmitted to the originating process as the file descriptor returned from **open**(BA_OS) or **creat**(BA_OS).

If the server does not respond to the **I_SENDFD** request, the stream that the **connld** module is pushed on becomes uni-directional because the server will not be able to retrieve any data off the stream until the **I_RECVFD** request is issued. If the server process exits before issuing the **I_RECVFD** request, the **open**(BA_OS) or **creat**(BA_OS) system calls will fail and return -1 to the originating process.

When the **connld** module is pushed onto a pipe, messages going back and forth through the pipe are ignored by **connld**.

ERRORS

On success, an open of **connld** returns 0. On failure, **errno** is set to the following values:

EINVAL A stream onto which **connld** is being pushed is not a pipe.

EINVAL The other end of the pipe onto which **connld** is being pushed is linked under a multiplexor.

EPIPE **connld**

NAME

devcon: console – system console interface

SYNOPSIS

`/dev/console`

DESCRIPTION

`/dev/console` is a generic name given to the system console. It is usually linked to a particular machine-dependent special file, and provides a basic I/O interface to the system console through the `termio` interface [see termio(BA_DEV)].

SEE ALSO

termio(BA_DEV), termios(BA_OS).

LEVEL

Level 1.

NAME

devnul: null – the null file

SYNOPSIS

`/dev/null`

DESCRIPTION

Data written on a null special file are discarded.

Read operations from a null special file always return 0 bytes.

Output of a command is written to the special file `/dev/null` when the command is executed for its side effects and not for its output.

LEVEL

Level 1.

NAME

devtty: tty – controlling terminal interface

SYNOPSIS

`/dev/tty`

DESCRIPTION

The file `/dev/tty` is, in each process, a synonym for the control terminal associated with the session of that process, if any. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected [see system(BA_OS)]. It can also be used for programs that demand the name of a file for output when typed output is desired and as an alternative to identifying what terminal is currently in use.

USAGE

Normally, application programs should not need to use this file interface. The standard input, standard output and standard error files should be used instead. These file are accessed through the `stdin`, `stdout` and `stderr stdio` interfaces, respectively.

SEE ALSO

system(BA_OS), termio(BA_DEV).

LEVEL

Level 1.

NAME

ldterm – standard STREAMS terminal line discipline module

DESCRIPTION

ldterm is a STREAMS module that provides most of the **termio**(BA_DEV) terminal interface. This module does not perform the low-level device control functions specified by flags in the **c_cflag** word of the **termio/termios** structure or by the **IGNBRK, IGNPAR, PARMRK**, or **INPCK** flags in the **c_iflag** word of the **termio/termios** structure; those functions must be performed by the driver or by modules pushed below the **ldterm** module. All other **termio/termios** functions are performed by **ldterm**; some of them, however, require the cooperation of the driver or modules pushed below **ldterm** and may not be performed in some cases. These include the **IXOFF** flag in the **c_iflag** word and the delays specified in the **c_oflag** word.

ldterm also handles EUC and multi-byte characters.

SEE ALSO

termio(BA_DEV), **termios**(BA_OS)

LEVEL

Level 1.

NAME

pckt – STREAMS Packet Mode module

DESCRIPTION

pckt is a STREAMS module that may be used with a pseudo terminal to packetize certain messages. The **pckt** module should be pushed [see **I_PUSH**, **streamio**(7)] onto the master side of a pseudo terminal.

SEE ALSO

getmsg(BA_OS), **ioctl**(BA_OS), **ldterm**(BA_DEV), **ptem**(BA_DEV), **streams**(BA_DEV), **termio**(BA_DEV)

LEVEL

Level 1.

NAME

ptem – STREAMS Pseudo Terminal Emulation module

DESCRIPTION

ptem is a

NAME

streamio – STREAMS **ioctl** commands

SYNOPSIS

```
#include <sys/types.h>
#include <stropts.h>

int ioctl (int fildes, int command, ... /* arg */);
```

DESCRIPTION

STREAMS **ioctl** commands are a subset of the **ioctl**() system calls which perform a variety of control functions on streams.

fildes is an open file descriptor that refers to a stream. *command* determines the control function to be performed as described below. *arg* represents additional information that is needed by this command. The type of *arg* depends upon the command, but it is generally an integer or a pointer to a *command*-specific data structure. The *command* and *arg* are interpreted by the stream head. Certain combinations of these arguments may be passed to a module or driver in the stream.

Since these STREAMS commands are a subset of **ioctl**, they are subject to the errors described there. In addition to those errors, the call will fail with **errno** set to **EINVAL**, without processing a control function, if the stream referenced by *fildes* is linked below a multiplexor, or if *command* is not a valid value for a stream.

Also, as described in **ioctl**, STREAMS modules and drivers can detect errors. In this case, the module or driver sends an error message to the stream head containing an error value.

Dynamically Loadable Modules

STREAMS modules and drivers may be dynamically loadable. If a dynamically loadable module or driver is accessed via an **open()** or an **I_PUSH** (streamio) and it is not currently present in memory, then it is automatically loaded as a side effect of the access. The loading process will bring the driver or module into memory and call its *load()* routine to initialize it. See modload(KE_OS), modadmin(AS_CMD).

Accessing STREAMS

A user process accesses STREAMS using the standard routines **open()** [see open(BA_OS)], **close()** [see close(BA_OS)], **read()**, **write()**, **ioctl()**, **pipe()** [see pipe(BA_OS)], **putmsg()**, **putpmsg()**, **getmsg()**, **getpmsg()**, and **poll()**. Refer to the detailed component definitions for these functions for general properties and errors.

ioctl() calls are used to perform control functions with the device associated with the file descriptor *fd*. The arguments *command* and *arg* are passed to the STREAMS file designated by *fd* and are interpreted by the stream head. Certain combinations of these arguments may be passed to a module or driver in the stream.

fd is an open file descriptor that refers to a stream. *command* determines the control function to be performed as described below. *arg* represents additional information that is needed by this command. The type of *arg* depends on the command, but it is generally an integer or a pointer to a command-specific data structure.

Since these STREAMS commands are a subset of **ioctl()**, they are subject to the errors described there. In addition to those errors, the call will fail with **errno** set to **EINVAL**, without processing a control function, if the stream referenced by *fd* is linked below a multiplexer, or if *command* is not a valid value for a stream.

STREAMS modules and drivers can detect errors, sending an error message to the stream head, thus causing subsequent system calls to fail and set **errno** to the value specified in the message. In addition, STREAMS modules and drivers can elect to fail a particular **ioctl()** request alone by sending a negative acknowledgement message to the stream head. This causes just the pending **ioctl()** request to fail and set **errno** to the value specified in the message.

ioctl() calls have the format:

```
int ioctl(int fd, int command, int arg);
```

The **ioctl()** commands applicable to STREAMS and their arguments are described below. Unless specified, the return value from **ioctl()** is **0** on success and **–1** on failure with **errno** set as indicated. **errno** will be set to **EINVAL** for any of the following **ioctl()** calls if the stream is linked below a multiplexer.

To use **ioctl()**, the lines

```
#include <sys/types.h>
#include <stropts.h>
```

must be included in the user program.

Command Functions

The following **ioctl** commands, with error values indicated, are applicable to all STREAMS files:

I_PUSH Pushes the module whose name is pointed to by *arg* onto the top of the current stream, just below the stream head. If the stream is a pipe, the module will be inserted between the stream heads of both ends of the pipe. It then calls the open routine of the newly-pushed module. On failure, **errno** is set to one of the following values:

- **EINVAL** Invalid module name.
- **ENXIO** Open routine of new module failed.
- **ENXIO** Hangup received on *fildes*.
- **ENOLOAD** failure in loading a loadable exec module

I_POP Removes the module just below the stream head of the stream pointed to by *fildes*. To remove a module from a pipe requires that the module was pushed on the side it is being removed from. *arg* should be 0 in an **I_POP** request. On failure, **errno** is set to one of the following values:

- **EINVAL** No module present in the stream.
- **ENXIO** Hangup received on *fildes*.

I_LOOK Retrieves the name of the module just below the stream head of the stream pointed to by *fildes*, and places it in a null terminated character string pointed at by *arg*. The buffer pointed to by *arg* should be at least **FMNAMESZ**+1 bytes long. A **#include <sys/conf.h>**

declaration is required. On failure, **errno** is set to one of the following values:

EINVAL
No module present in stream.

I_FLUSH This request flushes all input and/or output queues, depending on the value of *arg*. Valid *arg* values are:

FLUSHR Flush read queues.

FLUSHW Flush write queues.

FLUSHRW Flush read and write queues.

If a pipe or FIFO does not have any modules pushed, the read queue of the stream head on either end is flushed depending on the value of *arg*.

If **FLUSHR** is set and *fildes* is a pipe, the read queue for that end of the pipe is flushed and the write queue for the other end is flushed. If *fildes* is a FIFO, both queues are flushed.

If **FLUSHW** is set and *fildes* is a pipe and the other end of the pipe exists, the read queue for the other end of the pipe is flushed and the write queue for this end is flushed. If *fildes* is a FIFO, both queues of the FIFO are flushed.

If **FLUSHRW** is set, all read queues are flushed, that is, the read queue for the FIFO and the read queue on both ends of the pipe are flushed.

Correct flush handling of a pipe or FIFO with modules pushed is achieved via the **pipemod** module. This module should be the first module pushed onto a pipe so that it is at the midpoint of the pipe itself.

On failure, **errno** is set to one of the following values:

EAGAIN Unable to allocate buffers for flush message due to insufficient STREAMS memory resources.

EINVAL Invalid *arg* value.

ENXIO Hangup received on *fildes*.

I_FLUSHBAND Flushes a particular band of messages. *arg* points to a **bandinfo** structure that has the following members:

```
unsigned char   bi_pri;
int             bi_flag;
```

The **bi_flag** field may be one of **FLUSHR, FLUSHW**, or **FLUSHRW** as described earlier.

I_SETSIG Informs the stream head that the user wants the kernel to issue the **SIGPOLL** signal [see **signal**(BA_OS)] when a particular event has occurred on the stream associated with *fildes*. **I_SETSIG** supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the user should

be signaled. It is the bitwise-OR of any combination, except where noted, of the following constants:

S_INPUT Any message other than an **M_PCPROTO** has arrived on a stream head read queue. This event is maintained for compatibility with prior releases. This is set even if the message is of zero length.

S_RDNORM An ordinary (non-priority) message has arrived on a stream head read queue. This is set even if the message is of zero length.

S_RDBAND A priority band message (band > 0) has arrived on a stream head read queue. This is set even if the message is of zero length.

S_HIPRI A high priority message is present on the stream head read queue. This is set even if the message is of zero length.

S_OUTPUT The write queue just below the stream head is no longer full. This notifies the user that there is room on the queue for sending (or writing) data downstream.

S_WRNORM This event is the same as **S_OUTPUT**.

S_WRBAND A priority band greater than 0 of a queue downstream exists and is writable. This notifies the user that there is room on the queue for sending (or writing) priority data downstream.

S_MSG A STREAMS signal message that contains the **SIGPOLL** signal has reached the front of the stream head read queue.

S_ERROR An **M_ERROR** message has reached the stream head.

S_HANGUP An **M_HANGUP** message has reached the stream head.

S_BANDURG When used in conjunction with **S_RDBAND**, **SIGURG** is generated instead of **SIGPOLL** when a priority message reaches the front of the stream head read queue.

A user process may choose to be signaled only of high priority messages by setting the *arg* bitmask to the value **S_HIPRI**.

Processes that want to receive **SIGPOLL** signals must explicitly register to receive them using **I_SETSIG**. If several processes register to receive this signal for the same event on the same stream, each process will be signaled when the event occurs.

If the value of *arg* is zero, the calling process will be unregistered and will not receive further **SIGPOLL** signals. On failure, **errno** is set to one of the following values:

EINVAL *arg* value is invalid or *arg* is zero and process is not registered to receive the **SIGPOLL** signal.

EAGAIN Allocation of a data structure to store the signal request failed.

I_GETSIG Returns the events for which the calling process is currently registered to be sent a **SIGPOLL** signal. The events are returned as a bitmask pointed to by *arg*, where the events are those specified in the description of **I_SETSIG** above. On failure, **errno** is set to one of the following values:

EINVAL Process not registered to receive the **SIGPOLL** signal.

I_FIND Compares the names of all modules currently present in the stream to the name pointed to by *arg*, and returns 1 if the named module is present in the stream. It returns 0 if the named module is not present. On failure, **errno** is set to one of the following values:

EINVAL
arg does not contain a valid module name.

I_PEEK Allows a user to retrieve the information in the first message on the stream head read queue without taking the message off the queue. **I_PEEK** is analogous to **getmsg**() except that it does not remove the message from the queue. *arg* points to a **strpeek** structure which contains the following members:

```
struct strbuf     ctlbuf;
struct strbuf     databuf;
long              flags;
```

The **maxlen** field in the **ctlbuf** and **databuf strbuf** structures [see **getmsg**(BA_OS)] must be set to the number of bytes of control information and/or data information, respectively, to retrieve. **flags** may be set to **RS_HIPRI** or 0. If **RS_HIPRI** is set, **I_PEEK** will look for a high priority message on the stream head read queue. Otherwise, **I_PEEK** will look for the first message on the stream head read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the stream head read queue. It does not wait for a message to arrive. On return, **ctlbuf** specifies information in the control buffer, **databuf** specifies information in the data buffer, and **flags** contains the value **RS_HIPRI** or 0. On failure, **errno** is set to the following value:

I_SRDOPT Sets the read mode [see **read**(BA_OS)] using the value of the argument *arg*. Valid *arg* values are:

RNORM Byte-stream mode, the default.

RMSGD Message-discard mode.

RMSGN Message-nondiscard mode.

Setting both **RMSGD** and **RMSGN** is an error.
RMSGD and **RMSGN** override **RNORM**.

The bitwise inclusive **OR** of **RMSGD** and **RMSGN** will return **EINVAL**. The bitwise inclusive **OR** of **RMSRM** and either **RMSGD** or **RMSGN** will result in the other flag overridding **RNSRM** which is the default.

In addition, treatment of control messages by the stream head may be changed by setting the following flags in *arg*:

RPROTNORM Fail **read** with **EBADMSG** if a control message is at the front of the stream head read queue. This is the default behavior.

RPROTDAT Deliver the control portion of a message as data when a user issues **read**.

RPROTDIS Discard the control portion of a message, delivering any data portion, when a user issues a **read**.

On failure, **errno** is set to the following value:

EINVAL *arg* is not one of the above valid values.

EINVAL Both **RMSGD** and **RMSGN** are set.

I_GRDOPT Returns the current read mode setting in an **int** pointed to by the argument *arg*. Read modes are described in **read**(2).

I_NREAD Counts the number of data bytes in data blocks in the first message on the stream head read queue, and places this value in the location pointed to by *arg*. The return value for the command is the number of messages on the stream head read queue. For example, if zero is returned in *arg*, but the **ioctl** return value is greater than zero, this indicates that a zero-length message is next on the queue.

I_FDINSERT Creates a message from user specified buffer(s), adds information about another stream and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below.

arg points to a **strfdinsert** structure which contains the following members:

```
struct strbuf     ctlbuf;
struct strbuf     databuf;
long              flags;
int               fildes;
int               offset;
```

The **len** field in the **ctlbuf strbuf** structure [see **putmsg**(BA_OS)] must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. *fildes* in the **strfdinsert** structure specifies the file descriptor of the other stream. **offset**, which must be word-aligned, specifies the number of bytes

beyond the beginning of the control buffer where **I_FDINSERT** will store a pointer. This pointer will be the address of the read queue structure of the driver for the stream corresponding to **fildes** in the **strfdinsert** structure. The **len** field in the **databuf strbuf** structure must be set to the number of bytes of data information to be sent with the message or zero if no data part is to be sent.

flags specifies the type of message to be created. An ordinary (non-priority) message is created if **flags** is set to 0, a high priority message is created if **flags** is set to **RS_HIPRI**. For normal messages, **I_FDINSERT** will block if the stream write queue is full due to internal flow control conditions. For high priority messages, **I_FDINSERT** does not block on this condition. For normal messages, **I_FDINSERT** does not block when the write queue is full and **O_NONBLOCK** is set. Instead, it fails and sets **errno** to **EAGAIN**.

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks, regardless of priority or whether **O_NONBLOCK** has been specified. No partial message is sent. On failure, **errno** is set to one of the following values:

EAGAIN A non-priority message was specified, the **O_NONBLOCK** flag is set, and the stream write queue is full due to internal flow control conditions.

EAGAIN Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

EINVAL One of the following: **fildes** in the **strfdinsert** structure is not a valid, open stream file descriptor; the size of a pointer plus **offset** is greater than the **len** field for the buffer specified through **ctlptr**; **offset** does not specify a properly-aligned location in the data buffer; an undefined value is stored in **flags**.

ENXIO Hangup received on **fildes** of the **ioctl** call or **fildes** in the **strfdinsert** structure.

ERANGE The **len** field for the buffer specified through **databuf** does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module, or the **len** field for the buffer specified through **databuf** is larger than the maximum configured size of the data part of a message, or the **len** field for the buffer specified through **ctlbuf** is larger than the maximum configured size of the control part of a message.

I_FDINSERT can also fail if an error message was received by the stream head of the stream corresponding to **fildes** in the **strfdinsert** structure. In this case, **errno** will be set to the value in the message.

I_STR Constructs an internal STREAMS ioctl message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send user **ioctl** requests to downstream modules and drivers. It allows information to be sent with the **ioctl**, and will return to the user any information sent upstream by the downstream recipient. **I_STR** blocks until the sys-

An **I_STR** can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the stream head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the ioctl command sent downstream fails. For these cases, **I_STR** will fail with **errno** set to the value in the message.

I_SWROPT Sets the write mode using the value of the argument *arg*. Legal bit settings for *arg* are:

SNDZERO Send a zero-length message downstream when a write of 0 bytes occurs on pipes and FIFOs.

To not send a zero-length message when a write of 0 bytes occurs, this bit must not be set in *arg*.

On failure, **errno** may be set to the following value:

EINVAL *arg* is not the above valid value.

I_GWROPT Returns the current write mode setting, as described above, in the **int** that is pointed to by the argument *arg*.

I_SENDFD Requests the stream associated with *fildes* to send a message, containing a file pointer, to the stream head at the other end of a stream pipe. The file pointer corresponds to *arg*, which must be an open file descriptor.

I_SENDFD converts *arg* into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user ID and group ID associated with the sending process are also inserted. This message is placed directly on the read queue of the stream head at the other end of the stream pipe to which it is connected. On failure, **errno** is set to one of the following values:

EAGAIN The sending stream is unable to allocate a message block to contain the file pointer.

EAGAIN The read queue of the receiving stream head is full and cannot accept the message sent by **I_SENDFD**.

EBADF *arg* is not a valid, open file descriptor.

EINVAL *fildes* is not connected to a stream pipe.

ENXIO Hangup received on *fildes*.

I_RECVFD Retrieves the file descriptor associated with the message sent by an **I_SENDFD ioctl** over a stream pipe. *arg* is a pointer to a data buffer large enough to hold an **strrecvfd** data structure containing the following members:

```
int fd;
uid_t uid;
gid_t gid;
char fill[8];
```

fd is an integer file descriptor. **uid** and **gid** are the user ID and group ID, respectively, of the sending stream.

If **O_NONBLOCK** are clear [see **open**(BA_OS)] **I_RECVFD** will block until a message is present at the stream head. If **O_NONBLOCK** is set, **I_RECVFD** will fail with **errno** set to **EAGAIN** if no message is present at the stream head.

If the message at the stream head is a message sent by an **I_SENDFD**, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is placed in the **fd** field of the **strrecvfd** structure. The structure is copied into the user data buffer pointed to by *arg*.

On failure, **errno** is set to one of the following values:

EAGAIN A message is not present at the stream head read queue, and the **O_NONBLOCK** flag is set.

EBADMSG The message at the stream head read queue is not a message containing a passed file descriptor.

EFAULT *arg* points outside the allocated address space.

EMFILE **NOFILES** file descriptors are currently open.

ENXIO Hangup received on *fildes*.

EOVERFLOW **uid** or **gid** is too large to be stored in the structure pointed to by *arg*.

I_LIST Allows the user to list all the module names on the stream, up to and including the topmost driver name. If *arg* is **NULL**, the return value is the number of modules, including the driver, that are on the stream pointed to by *fildes*. This allows the user to allocate enough space for the module names. If *arg* is non-**NULL**, it should point to an **str_list** structure that has the following members:

```
int sl_nmods;
struct str_mlist   *sl_modlist;
```

The **str_mlist** structure has the following member:

```
char l_name[FMNAMESZ+1];
```

sl_nmods indicates the number of entries the user has allocated in the array. On success, the return value is 0, **sl_modlist** contains the list of module names, and the number of entries that have been filled into the **sl_modlist** array is found in the **sl_nmods** member. The number includes the number of modules, including the driver. On failure, **errno** may be set to one of the following values:

EINVAL The **sl_nmods** member is less than 1.

EAGAIN Unable to allocate buffers

I_ATMARK Allows the user to see if the current message on the stream head read queue is ''marked'' by some module downstream. *arg* determines how the checking is done when there may be multiple marked messages on the stream head read queue. The bitwise-OR of these flags is allowed. It may take the following values:

ANYMARK Check if the message is marked.

LASTMARK Check if the message is the last one marked on the queue.

If both **ANYMARK** and **LASTMARK** are set, **ANYMARK** supersedes **LASTMARK**.

The return value is 1 if the mark condition is satisfied and 0 otherwise. On failure, **errno** may be set to the following value:

EINVAL A value other than **(ANYMARK|LASTMARK)** is set in *arg*.

I_CKBAND Check if the message of a given priority band exists on the stream head read queue. This returns 1 if a message of a given priority exists, or –1 on error. *arg* should be an integer containing the value of the priority band in question. On failure, **errno** may be set to the following value:

EINVAL Invalid *arg* value.

I_GETBAND Returns the priority band of the first message on the stream head read queue in the integer referenced by *arg*. On failure, **errno** may be set to the following value:

ENODATA No message on the stream head read queue.

I_CANPUT Check if a certain band is writable. *arg* is set to the priority band in question. The return value is 0 if the priority band *arg* is flow controlled, 1 if the band is writable, or –1 on error. On failure, **errno** may be set to the following value:

EINVAL Invalid *arg* value.

I_SETCLTIME Allows the user to set the time the stream head will delay when a stream is closing and there is data on the write queues. Before closing each module and driver, the stream head will delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, data will be flushed. *arg* is a pointer to the number of milliseconds to delay, rounded up to the nearest valid value on the system. The default is fifteen seconds. On failure, **errno** may be set to the following value:

EINVAL Invalid *arg* value.

I_GETCLTIME Returns the close time delay in the long pointed by *arg*.

The following four commands are used for connecting and disconnecting multiplexed STREAMS configurations.

I_LINK Connects two streams, where *fildes* is the file descriptor of the stream connected to the multiplexing driver, and *arg* is the file descriptor of the stream connected to another driver. The stream designated by *arg* gets connected below the multiplexing driver. **I_LINK** requires the multiplexing driver to send an acknowledgement message to the stream head regarding the linking operation. This call returns a multiplexor ID number (an identifier used to disconnect the multiplexor, see **I_UNLINK**) on success, and a –1 on failure. On failure, **errno** is set to one of the following values:

ENXIO Hangup received on *fildes*.

ET

EINVAL *arg* is an invalid multiplexor ID number or *fildes* is not the stream on which the **I_LINK** that returned *arg* was performed.

An **I_UNLINK** can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, **I_UNLINK** will fail with **errno** set to the value in the message.

I_PLINK Connects two streams, where *fildes* is the file descriptor of the stream connected to the multiplexing driver, and *arg* is the file descriptor of the stream connected to another driver. The stream designated by *arg* gets connected via a persistent link below the multiplexing driver. **I_PLINK** requires the multiplexing driver to send an acknowledgement message to the stream head regarding the linking operation. This call creates a persistent link which can exist even if the file descriptor *fildes* associated with the upper stream to the multiplexing driver is closed. This call returns a multiplexor ID number (an identifier that may be used to disconnect the multiplexor, see **I_PUNLINK**) on success, and a –1 on failure. On failure, **errno** may be set to one of the following values:

ENXIO Hangup received on *fildes*.

ETIME Time out before acknowledgement message was received at the stream head.

EAGAIN Unable to allocate STREAMS storage to perform the **I_PLINK**.

EBADF *arg* is not a valid, open file descriptor.

EINVAL *fildes* does not support multiplexing.

EINVAL *arg* is not a stream or is already linked under a multiplexor.

EINVAL The specified link operation would cause a ‘‘cycle’’ in the resulting configuration; that is, if a given stream head is linked into a multiplexing configuration in more than one place.

An **I_PLINK** can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error on a hangup is received at the stream head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, **I_PLINK** will fail with **errno** set to the value in the message.

I_PUNLINK Disconnects the two streams specified by *fildes* and *arg* that are connected with a persistent link. *fildes* is the file descriptor of the stream connected to the multiplexing driver. *arg* is the multiplexor ID number that was returned by **I_PLINK** when a stream was linked below the multiplexing driver. If *arg* is **MUXID_ALL** then all streams

which are persistent links to *fildes* are disconnected. As in **I_PLINK**, this command requires the multiplexing driver to acknowledge the unlink. On failure, **errno** may be set to one of the following values:

ENXIO Hangup received on *fildes*.

ETIME Time out before acknowledgement message was received at the stream head.

EAGAIN Unable to allocate buffers for the acknowledgement message.

EINVAL Invalid multiplexor ID number.

EINVAL *fildes* is the file descriptor of a pipe or FIFO.

An **I_PUNLINK** can also fail while waiting for the multiplexing driver to acknowledge the link request if a message indicating an error or a hangup is received at the stream head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, **I_PUNLINK** will fail with **errno** set to the value.

Return Values

Unless specified otherwise above, **ioctl** returns 0 on success and –1 on failure and sets **errno** as indicated in the message.

SEE ALSO

close(BA_OS), **fcntl**(BA_OS), **getmsg**(BA_OS), **modadmin**(AS_CMD), **modload**(KE_OS), **open**(BA_OS), **poll**(BA_OS), **putmsg**(BA_OS), **read**(BA_OS), **signal**(BA_OS),

LEVEL

Level 1.

NAME

termio: ioctl – general terminal interface

SYNOPSIS

```
#include <termio.h>

ioctl(int fildes, int request, struct termio *arg);

ioctl(int fildes, int request, int arg);

#include <termios.h>

ioctl(int fildes, int request, struct termios *arg);
```

DESCRIPTION

System V supports a general interface for asynchronous communications ports that is hardware-independent. The user interface to this functionality is via function calls (the preferred interface) described in termios(BA_OS) or `ioctl()` commands described in this section. This section also discusses the common features of the terminal subsystem which are relevant with both user interfaces.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open terminal files; they are opened by the system and become a user's standard input, output, and error files. The very first terminal file opened by the session leader, which is not already associated with a session, becomes the control-terminal for that session. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a `fork()` [see fork(BA_OS)]. A process can break this association by changing its session using `setsid()` [see setsid(BA_OS)].

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the character input buffers of the system become completely full, which is rare (*e.g.*, if the number of characters in the line discipline buffer exceeds `{MAX_CANON}` and `IMAXBEL` [see below] is not set), or when data on the driver's input queue exceeds `{MAX_INPUT}` input characters that have not yet been read by some program. When the input limit is reached, all the characters saved in the buffer up to that point are thrown away without notice.

Session Management (Job Control)

A control terminal will distinguish one of the process groups in the session associated with it to be the foreground process group. All other process groups in the session are designated as background process groups. This foreground process group plays a special role in handling signal-generating input characters, as discussed below. By default, when a controlling terminal is allocated, the controlling process' process group is assigned as foreground process group.

Background process groups in the controlling process' session are subject to a job control line discipline when they attempt to access their controlling terminal. Typically, they will be sent a signal that will cause them to stop, unless they have made other arrangements. An exception is made for members of orphaned process group, process groups which do not have a member with a parent in another process group that is in the same session and therefore shares the same controlling terminal. When these processes attempt to access their controlling terminal, they will

return errors, since there is no process to continue them if they should stop.

If a member of a background process group attempts to read its controlling terminal, its process group will be sent a SIGTTIN signal, which will normally cause the members of that process group to stop. If, however, the process is ignoring or holding SIGTTIN, or is a member of an orphaned process group, the read will fail with errno set to EIO, and no signal will be sent.

If a member of a background process group attempts to write its controlling terminal and the TOSTOP bit is set in the c_lflag field, its process group will be sent a SIGTTOU signal, which will normally cause the members of that process group to stop. If, however, the process is ignoring or holding SIGTTOU, the write will succeed. If the process is not ignoring or holding SIGTTOU and is a member of an orphaned process group, the write will fail with *errno* set to EIO, and no signal will be sent.

If TOSTOP is set and a member of a background process group attempts to ioctl() its controlling terminal, and that ioctl() will modify terminal parameters (*e.g.*, TCSETA, TCSETAW, TCSETAF or TIOCSPGRP), its process group will be sent a SIGTTOU signal, which will normally cause the members of that process group to stop. If, however, the process is ignoring or holding SIGTTOU, the ioctl() will succeed. If the process is not ignoring or holding SIGTTOU and is a member of an orphaned process group, the write will fail with *errno* set to EIO, and no signal will be sent.

Canonical mode input processing

Normally, terminal input is processed in units of lines. A line is delimited by a newline (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not necessary, however, to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. The ERASE character (by default, the # character) erases the last character typed. The WERASE character (CRTL-W) erases the last ''word'' typed in the current input line (but not any preceding spaces or tabs). A ''word'' is defined as a sequence of non-blank characters, with tabs counted as blanks. Neither ERASE nor WERASE will erase beyond the beginning of the line. The KILL character (by default, the @ character) kills (deletes) the entire input line, and optionally outputs a newline character. All these characters operate on a key stroke basis, independent of any backspacing or tabbing that may have been done. The REPRINT character (CTRL-R) prints a newline followed by all characters that have not been read. Reprinting also occurs automatically if characters that would normally be erased from the screen are fouled by program output. The characters are reprinted as if they were being echoed; consequencely, if ECHO is not set, they are not printed.

The ERASE and KILL characters may be entered literally by preceding them with the escape character (\ ). In this case, the escape character is not read. The erase and kill characters may be changed.

Non-canonical mode input processing

In non-canonical mode input processing, input characters are not assembled into lines, and erase and kill processing does not occur. The MIN and TIME values are used to determine how to process the characters received.

MIN represents the minimum number of characters that should be received when the read is satisfied (*i.e.*, when the characters are returned to the user). TIME is a timer of 0.10-second granularity that is used to timeout bursty and short-term data transmissions. The four possible values for MIN and TIME and their interactions are described below.

Case A: MIN > 0, TIME > 0

In this case, TIME serves as an intercharacter timer and is activated after the first character is received. Since it is an intercharacter timer, it is reset after a character is received. The interaction between MIN and TIME is as follows: as soon as one character is received, the intercharacter timer is started. If MIN characters are received before the intercharacter timer expires (note that the timer is reset upon receipt of each character), the read is satisfied. If the timer expires before MIN characters are received, the characters received to that point are returned to the user. Note that if TIME expires, at least one character will be returned because the timer would not have been enabled unless a character was received. In this case (MIN > 0, TIME > 0), the read sleeps until the MIN and TIME mechanisms are activated by the receipt of the first character. If the number of characters read is less than the number of characters available, the timer is not reactivated and the subsequent read is satisfied immediately.

Case B: MIN > 0, TIME = 0

In this case, since the value of TIME is zero, the timer plays no role and only MIN is significant. A pending read is not satisfied until MIN characters are received (the pending read sleeps until MIN characters are received). A program that uses this case to read record based terminal I/O may block indefinitely in the read operation.

Case C: MIN = 0, TIME > 0

In this case, since MIN = 0, TIME no longer represents an intercharacter timer: it now serves as a read timer that is activated as soon as a read is done. A read is satisfied as soon as a single character is received or the read timer expires. Note that, in this case, if the timer expires, no character is returned. If the timer does not expire, the only way the read can be satisfied is if a character is received. In this case, the read will not block indefinitely waiting for a character; if no character is received within TIME*.10 seconds after the read is initiated, the read returns with zero characters.

Case D: MIN = 0, TIME = 0

In this case, return is immediate. The minimum of either the number of characters requested or the number of characters currently available is returned without waiting for more characters to be input.

Comparison of the different cases of MIN, TIME interaction

Some points to note about MIN and TIME:

1. In the following explanations, note that the interactions of MIN and TIME are not symmetric. For example, when MIN > 0 and TIME = 0, TIME has no effect. However, in the opposite case, where MIN = 0 and TIME > 0, both MIN and TIME play a role in that MIN is satisfied with the receipt of a single character.
2. Also note that in case A (MIN > 0, TIME > 0), TIME represents an intercharacter timer, whereas in case **C** (TIME = 0, TIME > 0), TIME represents a read timer.

These two points highlight the dual purpose of the MIN/TIME feature. Cases A and B, where MIN > 0, exist to handle burst mode activity (*e.g.*, file transfer programs), where a program would like to process at least MIN characters at a time. In case A, the intercharacter timer is activated by a user as a safety measure; in case B, the timer is turned off.

Cases **C** and D exist to handle single character, timed transfers. These cases are readily adaptable to screen-based applications that need to know if a character is present in the input queue before refreshing the screen. In case **C**, the read is timed, whereas in case D, it is not.

Another important note is that MIN is always just a minimum. It does not denote a record length. For example, if a program does a read of 20 bytes, MIN is 10, and 25 characters are present, then 20 characters will be returned to the user.

Writing characters

When one or more characters are written, they are transmitted to the terminal as soon as previously written characters have finished typing. Input characters are echoed as they are typed if echoing has been enabled. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue is drained down to some threshold, the program is resumed.

Special characters

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (Rubout or ASCII DEL) generates a SIGINT signal, which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed upon location. [See signal(BA_OS).]

QUIT (CTRL-| or ASCII FS) generates a SIGQUIT signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called core) will be created in the current working directory.

ERASE (#) erases the preceding character. It does not erase beyond the start of a line, as delimited by a NL, EOF, EOL, or EOL2 character.

WERASE (CTRL-W or ASCII ETX) erases the preceding ''word''. It does not erase beyond the start of a line, as delimited by a NL, EOF, EOL, or EOL2 character.

KILL (@) deletes the entire line, as delimited by a NL, EOF, EOL, or EOL2 character.

REPRINT (CTRL-R or ASCII DC2) reprints all characters, preceded by a newline, that have not been read.

EOF (CTRL-D or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a newline, and the EOF is discarded. Thus, if no characters are waiting (*i.e.*, the EOF occurred at the beginning of a line) zero characters are passed back, which is the standard end-of-file indication. Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

NL (ASCII LF) is the normal line delimiter. It cannot be changed or escaped.

EOL (ASCII NULL) is an additional line delimiter, like NL. It is not normally used.

EOL2 is another additional line delimiter.

SUSP (CTRL-Z or ASCII SUB) It generates a SIGTSTP signal, which stops all processes in the foreground process group for that terminal.

DSUSP (CTRL-Y or ASCII EM) It generates a SIGTSTP signal as SUSP does, but the signal is sent when a process in the foreground process group attempts to read the DSUSP character, rather than when it is typed.

STOP (CTRL-S or ASCII DC3) can be used to suspend output temporarily. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (CTRL-Q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read.

DISCARD (CTRL-O or ASCII SI) causes subsequent output to be discarded until another DISCARD character is typed, more input arrives, or the condition is cleared by a program.

LNEXT (CTRL-V or ASCII SYN) causes the special meaning of the next character to be ignored; this works for all the special characters mentioned above. It allows characters to be input that would otherwise be interpreted by the system (*e.g.*, KILL, QUIT).

character, in which case no special function is done. Any of the special characters may be preceded by the LNEXT character, in which case no special function is done.

Modem disconnect

When a modem disconnect is detected, a SIGHUP signal is sent to the terminal's controlling process. Unless other arrangements have been made, this signal causes the process to terminate. If SIGHUP is ignored or caught, any subsequent read returns with an end-of-file indication until the terminal is closed.

Processes in background process groups that attempt to access the controlling terminal after modem disconnect while the terminal is still allocated to the session will receive appropriate SIGTTOU and SIGTTIN signals. Unless other arrangements have been made, this signal causes the processes to stop.

The controlling terminal will remain in this state until it is reinitialized with a successful open by the controlling process, or deallocated by the controlling process.

Terminal parameters

The parameters that control the behavior of devices and modules providing the termios interface are specified by the termios structure defined by

Input modes

The c_iflag field describes the basic terminal input control:

IGNBRK Ignore break condition.
If IGNBRK is set, a break condition (a character framing error with data all zeros) detected on input is ignored, that is, not put on the input queue and therefore not read by any process.

BRKINT Signal interrupt on break.
If IGNBRK is not set and BRKINT is set, the break condition shall flush the input and output queues and if the terminal is the controlling terminal of a foreground process group, the break condition generates a single SIGINT signal to that foreground process group. If neither IGNBRK nor BRKINT is set, a break condition is read as a single ASCII NULL character (´\0´), or if PARMRK is set, as ´\377´, ´\0´, ´\0´.

IGNPAR Ignore characters with parity errors.
If IGNPAR is set, a byte with framing or parity errors (other than break) is ignored.

PARMRK Mark parity errors.
If PARMRK is set, and IGNPAR is not set, a byte with a framing or parity error (other than break) is given to the application as the three-character sequence: ´\377´, ´\0´, *X*, where *X* is the data of the byte received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of ´\377´ is given to the application as ´\377´, ´\377´. If neither IGNPAR nor PARMRK is set, a framing or parity error (other than break) is given to the application as a single ASCII NULL character (´\0´).

INPCK Enable input parity check.
If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors. Note that whether input parity checking is enabled or disabled is independent of whether parity detection is enabled or disabled. If parity detection is enabled but input parity checking is disabled, the hardware to which the terminal is connected will recognize the parity bit, but the terminal special file will not check whether this is set correctly or not.

ISTRIP Strip character.
If ISTRIP is set, valid input characters are first stripped to seven bits, otherwise all eight bits are processed.

INLCR Map NL to CR on input.
If INLCR is set, a received NL character is translated into a CR character.

IGNCR Ignore CR.
If IGNCR is set, a received CR character is ignored (not read).

ICRNL Map CR to NL on input.
If ICRNL is set, a received CR character is translated into a NL character.

IUCLC Map upper-case to lower-case on input.
If IUCLC is set, a received upper case, alphabetic character is translated into the corresponding lower case character.

IXON Enable start/stop output control.
If IXON is set, start/stop output control is enabled. A received STOP character suspends output and a received START character restarts output. The STOP and START characters will not be read, but will merely perform flow control functions.

IXANY Enable any character to restart output.
If IXANY is set, any input character restarts output that has been suspended.

IXOFF Enable start/stop input control.
If IXOFF is set, the system transmits a STOP character when the input queue is nearly full, and a START character when enough input has been read so that the input queue is nearly empty again.

IMAXBEL Echo BEL on input line too long.
If IMAXBEL is set, the ASCII BEL character is echoed if the input stream overflows. Further input is not stored, but any input already present in the input stream is not disturbed. If IMAXBEL is not set, no BEL character is echoed, and all input present in the input queue is discarded if the input stream overflows.

The initial input control value is BRKINT, ICRNL, IXON, ISTRIP.

Output modes

The c_oflag field specifies the system treatment of output:

OPOST Post-process output.
If OPOST is set, output characters are post-processed as indicated by the remaining flags; otherwise, characters are transmitted without change.

OLCUC Map lower case to upper on output.
If OLCUC is set, a lower case alphabetic character is transmitted as the corresponding upper case character. This function is often used in conjunction with IUCLC.

ONLCR Map NL to CR-NL on output.
If ONLCR is set, the NL character is transmitted as the CR-NL character pair.

OCRNL Map CR to NL on output.
If OCRNL is set, the CR character is transmitted as the NL character.

ONOCR No CR output at column 0.
If ONOCR is set, no CR character is transmitted when at column 0 (first position).

ONLRET NL performs CR function.
If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer is set to 0 and the delays specified for CR are used. Otherwise, the NL character is assumed to do just the

linefeed function; the column pointer remains unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

OFILL Use fill characters for delay.
If OFILL is set, fill characters are transmitted for delay instead of a timed delay. This is useful for high baud rate terminals that need only a minimal delay.

OFDEL Fill is DEL, else NULL.
If OFDEL is set, the fill character is DEL; otherwise it is NULL.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases, a value of 0 indicates no delay.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

The actual delays depend on line speed and system load.

NLDLY Newline delay lasts about 0.10 seconds.
If ONLRET is set, the carriage-return delays are used instead of the new-line delays.

If OFILL is set, two fill characters are transmitted.

Select new-line delays.
NL0 New-Line character type 0
NL1 New-Line character type 1

CRDLY Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds.

If OFILL is set, delay type 1 transmits two fill characters, and type 2 transmits four fill characters.

Select carriage-return delays:
CR0 Carriage-return delay type 0
CR1 Carriage-return delay type 1
CR2 Carriage-return delay type 2
CR3 Carriage-return delay type 3

TABDLY Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces.

If OFILL is set, two fill characters are transmitted for any delay.

Select horizontal tab delays or tab expansion:
TAB0 Horizontal-tab delay type 0
TAB1 Horizontal-tab delay type 1
TAB2 Horizontal-tab delay type 2
TAB3 Expand tabs to spaces.
XTABS Expand tabs to spaces.

BSDLY Backspace delay lasts about 0.05 seconds.

If OFILL is set, one fill character is transmitted.

Select backspace delays:
BS0 Backspace delay type 0
BS1 Backspace delay type 1

VTDLY Vertical-tab delay lasts about 2.0 seconds.

Select vertical tab delays:
VT0 Vertical-tab delay type 0
VT1 Vertical-tab delay type 1

FFDLY Form-feed delay lasts about 2.0 seconds.

Select form feed delays:
FF0 Form-feed delay type 0
FF1 Form-feed delay type 1

The initial output control value is OPOST, ONLCR, TAB3.

Control modes

The c_cflag field describes the hardware control of the terminal:

CBAUD The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal is not asserted. Normally, this disconnects the line. If the CIBAUD bits are not zero, they specify the input baud rate, with the CBAUD bits specifying the output baud rate; otherwise, the output and input baud rates are both specified by the CBAUD bits. The values for the CIBAUD bits are the same as the values for the CBAUD bits, shifted left IBSHIFT bits. For any particular hardware, impossible speed changes are ignored.

Baud rate:

B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
EXTA	External A
EXTB	External B

CSIZE The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any.

Character size:
CS5 5 bits
CS6 6 bits
CS7 7 bits
CS8 8 bits

CSTOPB Send two stop bits, else one
If CSTOPB is set, two stop bits are used; otherwise, one stop bit is used. For example, at 110 baud, two stops bits are required.

CREAD Enable receiver
If CREAD is set, the receiver is enabled. Otherwise, no characters are received.

PARENB Parity enable
If PARENB is set, parity generation and detection is enabled, and a parity bit is added to each character.

PARODD Odd parity, else even
If parity is enabled, the PARODD flag specifies odd parity if set; otherwise, even parity is used.

HUPCL Hang up on last close
If HUPCL is set, the line is disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal is not asserted.

CLOCAL Local line, else dial-up
If CLOCAL is set, then the effect of setting the baud rate to 0 is driver-dependent. If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control; otherwise, modem control is assumed.

CIBAUD Input baud rate, if different from output rate.

PAREXT Extended parity for mark and space parity.

The initial hardware control value after open is CS8, CREAD, HUPCL.

Local modes and line discipline

The c_lflag field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG Enable signals.
If ISIG is set, each input character is checked against the special control characters INTR, QUIT, and SUSP, STATUS, and DSUSP. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus, these special input functions are possible only if ISIG is set.

`ICANON` Canonical input (erase and kill processing).
If `ICANON` is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by `NL`, `EOF`, `EOL`, and `EOL2`. If `ICANON` is not set, read requests are satisfied directly from the input queue. A read is not satisfied until at least `MIN` characters have been received or the timeout value `TIME` has expired between characters. This allows fast bursts of input to be read efficiently while still allowing single character input. The time value represents tenths of seconds.

`XCASE` Canonical upper/lower presentation.
If `XCASE` is set, and if `ICANON` is set, an upper case letter is accepted on input by preceding it with a `\` character, and is output preceded by a `\` character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	**use:**
`	\´
|	\!
~	\^
{	\(
}	\)
\	\\

For example, `A` is input as `\a`, `\n` as `\\n`, and `\N` as `\\\n`.

`ECHO` Enable echo.
If `ECHO` is set, characters are echoed as received.

When `ICANON` is set, the following echo functions are possible:

`ECHOE` Echo erase character as `BS-SP-BS`.
If `ECHO` and `ECHOE` are set, and `ECHOPRT` is not set, the `ERASE` and `WERASE` characters are echoed as one or more `ASCII BS SP BS`, which clears the last character(s) from a `CRT` screen.

`ECHOK` Echo NL after kill character.
If `ECHOK` is set, and `ECHOKE` is not set, the `NL` character is echoed after the kill character to emphasize that the line is deleted. Note that an escape character (\) or an `LNEXT` character preceding the erase or kill character removes any special function.

`ECHONL` Echo NL.
If `ECHONL` is set, the `NL` character is echoed even if `ECHO` is not set. This is useful for terminals set to local echo (so called half-duplex).

`NOFLSH` Disable flush after interrupt or quit.
If `NOFLSH` is set, the normal flush of the input and output queues associated with the `INTR`, `QUIT`, and `SUSP` characters is not done.

`TOSTOP` Send `SIGTTOU` for background output.
If `TOSTOP` is set, the signal `SIGTTOU` is sent to a process that tries to write to its controlling terminal if it is not in the foreground process group for that terminal. This signal normally stops the process.

Otherwise, the output generated by that process is output to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output, if any.

ECHOCTL Echo control characters as ^*char*, delete as ^?.
If ECHOCTL is set, all control characters (characters with codes between 0 and 37 octal) other than ASCII TAB, ASCII NL, the START character, the STOP character, ASCII CR, and ASCII BS are echoed as ^*X*, where *X* is the character given by adding 100 octal to the code of the control character (so that the character with octal code 1 is echoed as ^A), and the ASCII DEL character, with code 177 octal, is echoed as ^?.

ECHOPRT Echo erase character as character erased.
If ECHO and ECHOPRT are set, the first ERASE and WERASE character in a sequence echoes as a backslash (\), followed by the characters being erased. Subsequent ERASE and WERASE characters echo the characters being erased, in reverse order. The next non-erase character causes a slash (/) to be typed before it is echoed.

ECHOKE BS-SP-BS erase entire line on line kill.
If ECHOKE is set, the kill character is echoed by erasing each character on the line from the screen (using the mechanism selected by ECHOE and ECHOPRT).

FLUSHO Output is being flushed.
If FLUSHO is set, data written to the terminal is discarded. This bit is set when the FLUSH character is typed. A program can cancel the effect of typing the FLUSH character by clearing FLUSHO.

PENDIN Retype pending input at next read or input character.
If PENDIN is set, any input that has not yet been read is reprinted when the next character arrives as input.

IEXTEN Enable extended (implementation-defined) functions. By default, IEXTEN is not set and processing of the following is disabled:

special characters WERASE, REPRINT, DISCARD and LNEXT;

local flags TOSTOP, ECHOCTL, ECHOPRT, ECHOKE, FLUSHO and PENDIN.

The initial line-discipline control value is ISIG, ICANON, ECHO, ECHOK.

Minimum and Timeout

The MIN and TIME values are described above under **Non-canonical mode input processing**. The initial value of MIN is 1, and the initial value of TIME is 0.

Terminal size

The number of lines and columns on the terminal's display is specified in the winsize structure defined by

```
unsigned   short   ws_row;     /* rows, in characters */
unsigned   short   ws_col;     /* columns, in characters */
unsigned   short   ws_xpixel;  /* horizontal size, in pixels*/
unsigned   short   ws_ypixel;  /* vertical size, in pixels*/
```

Termio structure

The System V `termio` structure is used by some `ioctl()`s; it is defined by `<sys/termio.h>` and includes the following members:

```
unsigned   short   c_iflag;    /* input modes */
unsigned   short   c_oflag;    /* output modes */
unsigned   short   c_cflag;    /* control modes */
unsigned   short   c_lflag;    /* local modes */
char               c_line;     /* line discipline */
unsigned   char    c_cc[NCC];  /* control chars */
```

The special control characters are defined by the array `c_cc`. The symbolic name `NCC` is the size of the control-character array and is also defined by `<termio.h>`. All space in the array (up to subscript `NCC`) is reserved or used as described below. The relative positions, symbolic subscript names, and typical default values for each function are as follows:

```
VINTR         DEL
VQUIT         FS
VERASE        #
VKILL         @
VEOF          EOT
VEOL          NUL
VEOL2         NUL
```

The calls that use the `termio` structure only affect the flags and control characters that can be stored in the `termio` structure; all other flags and control characters are unaffected.

Modem lines

On special files representing serial ports, the modem control lines supported by the hardware can be read, and the modem status lines supported by the hardware can be changed. The following modem control and status lines may be supported by a device; they are defined by `<sys/termios.h>`:

`TIOCM_LE`	line enable
`TIOCM_DTR`	data terminal ready
`TIOCM_RTS`	request to send
`TIOCM_ST`	secondary transmit
`TIOCM_SR`	secondary receive
`TIOCM_CTS`	clear to send
`TIOCM_CAR`	carrier detect
`TIOCM_RNG`	ring
`TIOCM_DSR`	data set ready

`TIOCM_CD` is a synonym for `TIOCM_CAR`, and `TIOCM_RI` is a synonym for `TIOCM_RNG`. Not all of these are necessarily supported by any particular device; check the manual page for the device in question.

ioctls

The `ioctl()`s supported by devices and `STREAMS` modules providing the `termios` interface are listed below. Some calls may not be supported by all devices or modules. The functionality provided by these calls is also available through the preferred function call interface specified on termios(BA_OS).

`TCGETS` The argument is a pointer to a `termios` structure. The current terminal parameters are fetched and stored into that structure.

`TCSETS` The argument is a pointer to a `termios` structure. The current terminal parameters are set from the values stored in that structure. The change is immediate.

`TCSETSW` The argument is a pointer to a `termios` structure. The current terminal parameters are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted. This form should be used when changing parameters that affect output.

`TCSETSF` The argument is a pointer to a `termios` structure. The current terminal parameters are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted; all characters queued for input are discarded and then the change occurs.

`TCGETA` The argument is a pointer to a `termio` structure. The current terminal parameters are fetched, and those parameters that can be stored in a `termio` structure are stored into that structure.

`TCSETA` The argument is a pointer to a `termio` structure. Those terminal parameters that can be stored in a `termio` structure are set from the values stored in that structure. The change is immediate.

`TCSETAW` The argument is a pointer to a `termio` structure. Those terminal parameters that can be stored in a `termio` structure are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted. This form should be used when changing parameters that affect output.

`TCSETAF` The argument is a pointer to a `termio` structure. Those terminal parameters that can be stored in a `termio` structure are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted; all characters queued for input are discarded and then the change occurs.

`TCSBRK` The argument is an `int` value. Wait for the output to drain. If the argument is 0, then send a break (zero valued bits for 0.25 seconds).

`TCXONC` Start/stop control. The argument is an `int` value. If the argument is 0, suspend output; if 1, restart suspended output; if 2, suspend input; if 3, restart suspended input.

`TCFLSH` The argument is an `int` value. If the argument is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

`TIOCGPGRP` The argument is a pointer to a `pid_t`. Set the value of that `pid_t` to the process group `ID` of the foreground process group associated with the terminal. [See termios(BA_OS) for a description of `tcgetpgrp`.]

`TIOCSPGRP` The argument is a pointer to a `pid_t`. Associate the process group whose process group ID is specified by the value of that `pid_t` with the terminal. The new process group value must be in the range of valid process group ID values. Otherwise, the error `EPERM` is returned. [See termios(BA_OS) for a description of `tcsetpgrp`.]

`TIOCGSID` The argument is a pointer to an `pid_t`. The session ID of the terminal is fetched and stored in the `pid_t`.

`TIOCGWINSZ` The argument is a pointer to a `winsize` structure. The terminal driver's notion of the terminal size is stored into that structure.

`TIOCSWINSZ` The argument is a pointer to a `winsize` structure. The terminal driver's notion of the terminal size is set from the values specified in that structure. If the new sizes are different from the old sizes, a `SIGWINCH` signal is set to the process group of the terminal.

`TIOCMBIS` The argument is a pointer to an `int` whose value is a mask containing modem control lines to be turned on. The control lines whose bits are set in the argument are turned on; no other control lines are affected.

`TIOCMBIC` The argument is a pointer to an `int` whose value is a mask containing modem control lines to be turned off. The control lines whose bits are set in the argument are turned off; no other control lines are affected.

`TIOCMGET` The argument is a pointer to an `int`. The current state of the modem status lines is fetched and stored in the `int` pointed to by *arg*.

`TIOCMSET` The argument is a pointer to an `int` containing a new set of modem control lines. The modem control lines are turned on or off, depending on whether the bit for that mode is set or clear.

FILES

files in or under `/dev`

SEE ALSO

fork(BA_OS), ioctl(BA_OS), setsid(BA_OS), signal(BA_OS), streams(BA_DEV), termios(BA_OS).

LEVEL

Level 1.

NAME

termiox – extended general terminal interface

SYNOPSIS

```
#include <termiox.h>

ioctl(int fildes, int request, struct termiox *arg);
```

DESCRIPTION

The extended general terminal interface supplements the termio(BA_DEV) general terminal interface by adding support for asynchronous hardware flow control, isochronous flow control and clock modes, and local implementations of additional asynchronous features. Some systems may not support all of these capabilities because of either hardware or software limitations. Other systems may not permit certain functions to be disabled. In these cases, the appropriate bits will be ignored. If the capabilities can be supported, the interface described here must be used.

Hardware Flow Control Modes

Hardware flow control supplements the `termio` `IXON`, `IXOFF` and `IXANY` [see termio(BA_DEV)] character flow control. Character flow control occurs when one device controls the data transfer of another device by the insertion of control characters in the data stream between devices. Hardware flow control occurs when one device controls the data transfer of another device using electrical control signals on wires (circuits) of the asynchronous interface. Isochronous hardware flow control occurs when one device controls the data transfer of another device by asserting or removing the transmit clock signals of that device. Character flow control and hardware flow control may be simultaneously set.

In asynchronous, full duplex applications, the use of the Electronics Industries Association's EIA-232-D Request To Send (RTS) and Clear to Send (CTS) circuits is the preferred method of hardware flow control. An interface to other hardware flow control methods is included to provide a standard interface to these existing methods.

The EIA-232-D standard specified only unidirectional hardware flow control - the Data Circuit-terminating Equipment or Data Communications Equipment (DCE) indicates to the Data Terminal Equipment (DTE) to stop transmitting data. The `termiox` interface allows both unidirectional and bidirectional hardware flow control; when bidirectional flow control is enabled, either the DCE or DTE can indicate to each other to stop transmitting data across the interface. Note: It is assumed that the asynchronous port is configured as a DTE. If the connected device is also a DTE and not a DCE, then DTE to DTE (*e.g.*, terminal or printer connected to computer) hardware flow control is possible by using a null modem to interconnect the appropriate data and control circuits.

Clock Modes

Isochronous communication is a variation of asynchronous communication whereby two communicating devices may provide transmit and/or receive clock to each other. Incoming clock signals can be taken from the baud rate generator on the local isochronous port controller, from CCITT V.24 circuit 114, Transmitter Signal Element Timing - DCE source (EIA-232-D pin 15), or from CCITT V.24 circuit 115, Receiver Signal Element Timing - DCE source (EIA-232-D pin 17). Outgoing clock signals can be sent on CCITT V.24 circuit 113, Transmitter Signal Element

Timing - DTE source (EIA-232-D pin 24), sent on CCITT V.24 circuit 128, Receiver Signal Element Timing - DTE source (no EIA-232-D pin), or not sent at all.

In terms of clock modes, traditional asynchronous communication is implemented simply by using the local baud rate generator as the incoming transmit and receive clock source and not outputting any clock signals.

Terminal Parameters

The parameters that control the behavior of devices providing the `termiox` interface are specified by the `termiox` structure, defined in the `<sys/termiox.h>` header file. Several `ioctl()` system calls [see ioctl(BA_OS)] that fetch or change these parameters use the `termiox` structure, which contains the following members:

```
unsigned short x_hflag;     /* hardware flow control modes */
unsigned short x_cflag;     /* clock modes */
unsigned short x_rflag[NFF];/* reserved modes */
unsigned short x_sflag;     /* spare local modes */
```

The `x_hflag` field describes hardware flow control modes:

`RTSXOFF`	0000001	Enable RTS hardware flow control on input.
`CTSXON`	0000002	Enable CTS hardware flow control on output.
`DTRXOFF`	0000004	Enable DTR hardware flow control on input.
`CDXON`	0000010	Enable CD hardware flow control on output.
`ISXOFF`	0000020	Enable isochronous hardware flow control on input.

The EIA-232-D DTR and CD circuits are used to establish a connection between two systems. The RTS circuit is also used to establish a connection with a modem. Thus, both DTR and RTS are activated when an asynchronous port is opened. If DTR is used for hardware flow control, then RTS must be used for connectivity. If CD is used for hardware flow control, then CTS must be used for connectivity. Thus, RTS and DTR (or CTS and CD) cannot both be used for hardware flow control at the same time. Other mutual exclusions may exist, such as the simultaneous setting of the `termio HUPCL` and the `termiox DTRXOFF` bits, which use the DTE Ready line for different functions.

Variations of different hardware flow control methods may be selected by setting the the appropriate bits. For example, bidirectional RTS/CTS flow control is selected by setting both the `RTSXOFF` and `CTSXON` bits and bidirectional DTR/CTS flow control is selected by setting both the `DTRXOFF` and `CTSXON`. Modem control or unidirectional CTS hardware flow control is selected by setting only the `CTSXON` bit.

As previously mentioned, it is assumed that the local asynchronous port (*e.g.*,computer) is configured as a DTE. If the connected device (*e.g.*, printer) is also a DTE, it is assumed that the device is connected to the computer's asynchronous port via a null modem that swaps control circuits (typically RTS and CTS). The connected DTE drives RTS and the null modem swaps RTS and CTS so that the remote RTS is received as CTS by the local DTE. In the case that `CTSXON` is set for hardware flow control, a printer's lowering of its RTS would cause CTS seen by the computer to be lowered. Output to the printer is suspended until the the printer's

raising of its RTS, which would cause CTS seen by the computer to be raised.

If RTSXOFF is set, the Request to Send (RTS) circuit (line) will be raised, and if the asynchronous port needs to have its input stopped, it will lower the Request to Send (RTS) line. If the RTS line is lowered, it is assumed that the connected device will stop its output until RTS is raised.

If CTSXON is set, output will occur only if the Clear To Send (CTS) circuit (line) is raised by the connected device. If the CTS line is lowered by the connected device, output is suspended until CTS is raised.

If DTRXOFF is set, the DTE Ready (DTR) circuit (line) will be raised, and if the asynchronous port needs to have its input stopped, it will lower the DTE Ready (DTR) line. If the DTR line is lowered, it is assumed that the connected device will stop its output until DTR is raised.

If CDXON is set, output will occur only if the Received Line Signal Detector (CD) circuit (line) is raised by the connected device. If the CD line is lowered by the connected device, output is suspended until CD is raised.

If ISXOFF is set, and if the isochronous port needs to have its input stopped, it will stop the outgoing clock signal. It is assumed that the connected device is using this clock signal to create its output. Transit and receive clock sources are programmed using the x_cflag fields. If the port is not programmed for external clock generation, ISXOFF is ignored. Output isochronous flow control is supported by appropriate clock source programming using the x_cflag field and enabled at the remote connected device.

The x_cflag field specifies the system treatment of clock modes.

```
XMTCLK     0000007  Transmit clock source:
XCIBRG     0000000  Get transmit clock from Internal
                    Baud Rate Generator.
XCTSET     0000001  Get transmit clock from Transmitter
                    Signal Element Timing (DCE source)
                    lead, CCITT V.24 circuit 114,
                    EIA-232-D pin 15.
XCRSET     0000002  Get transmit clock from Receiver
                    Signal Element Timing (DCE source)
                    lead, CCITT V.24 circuit 115,
                    EIA-232-D pin 17.
RCVCLK     0000070  Receive clock source:
RCIBRG     0000000  Get receive clock from Internal
                    Baud Rate Generator.
RCTSET     0000010  Get receive clock from Transmitter
                    Signal Element Timing (DCE source)
                    lead, CCITT V.24 circuit 114,
                    EIA-232-D pin 15.
RCRSET     0000020  Get receive clock from Receiver
                    Signal Element Timing (DCE source)
                    lead, CCITT V.24 circuit 115,
                    EIA-232-D pin 17.
TSETCLK    0000700  Transmitter Signal Element Timing
```

```
                    (DTE source) lead, CCITT V.24
                    circuit 113, EIA-232-D pin 24,
                    clock source:
TSETCOFF   0000000  TSET clock not provided.
TSETCRBRG  0000100  Output receive baud rate generator
                    on circuit 113.
TSETCTBRG  0000200  Output transmit baud rate generator
                    on circuit 113.
TSETCTSET  0000300  Output transmitter signal element
                    timing (DCE source) on circuit 113.
TSETCRSET  0000400  Output receiver signal element
                    timing (DCE source) on circuit 113.
RSETCLK    0007000  Receiver Signal Element Timing (DTE
                    source) lead, CCITT V.24 circuit 128,
                    no EIA-232-D pin, clock source:
RSETCOFF   0000000  RSET clock not provided.
RSETCRBRG  0001000  Output receive baud rate generator
                    on circuit 128.
RSETCTBRG  0002000  Output transmit baud rate generator
                    on circuit 128.
RSETCTSET  0003000  Output transmitter signal element
                    timing (DCE source) on circuit 128.
RSETCRSET  0004000  Output receiver signal element
                    timing (DCE source) on circuit 128.
```

If the XMTCLK field has a value of XCIBRG, the transmit clock is taken from the hardware internal baud rate generator, as in normal asynchronous transmission. If XMTCLK = XCTSET, the transmit clock is taken from the Transmitter Signal Element Timing (DCE source) circuit. If XMTCLK = XCRSET, the transmit clock is taken from the Receiver Signal Element Timing (DCE source) circuit.

If the RCVCLK field has a value of RCIBRG, the receive clock is taken from the hardware Internal Baud Rate Generator, as in normal asynchronous transmission. If RCVCLK = RCTSET, the receive clock is taken from the Transmitter Signal Element Timing (DCE source) circuit. If RCVCLK = RCRSET, the receive clock is taken from the Receiver Signal Element Timing (DCE source) circuit.

If the TSETCLK field has a value of TSETCOFF, the Transmitter Signal Element Timing (DTE source) circuit is not driven. If TSETCLK = TSETCRBRG, the Transmitter Signal Element Timing (DTE source) circuit is driven by the Receive Baud Rate Generator. If TSETCLK = TSETCTBRG, the Transmitter Signal Element Timing (DTE source) circuit is driven by the Transmit Baud Rate Generator. If TSETCLK = TSETCTSET, the Transmitter Signal Element Timing (DTE source) circuit is driven by the Transmitter Signal Element Timing (DCE source). If TSETCLK = TSETCRBRG, the Transmitter Signal Element Timing (DTE source) circuit is driven by the Receiver Signal Element Timing (DCE source).

If the RSETCLK field has a value of RSETCOFF, the Receiver Signal Element Timing (DTE source) circuit is not driven. If RSETCLK = RSETCRBRG, the Receiver Signal Element Timing (DTE source) circuit is driven by the Receive Baud Rate Generator. If RSETCLK = RSETCTBRG, the Receiver Signal Element Timing (DTE source) circuit is driven by the Transmit Baud Rate Generator. If RSETCLK =

RSETCTSET, the Receiver Signal Element Timing (DTE source) circuit is driven by the Transmitter Signal Element Timing (DCE source). If RSETCLK = RSETCRBRG, the Receiver Signal Element Timing (DTE source) circuit is driven by the Receiver Signal Element Timing (DCE source).

The x_rflag field is reserved for future interface definitions and should not be used by any implementations. The x_sflag field may be used by local implementations wishing to customize their terminal interface using the termiox ioctl() system calls.

IOCTLS

The ioctl() system calls have the form:

```
ioctl(fildes, command, arg)
struct termiox *arg;
```

The commands using this form are:

TCGETX The argument is a pointer to a termiox structure. The current terminal parameters are fetched and stored into that structure.

TCSETX The argument is a pointer to a termiox structure. The current terminal parameters are set from the values stored in that structure. The change is immediate.

TCSETXW The argument is a pointer to a termiox structure. The current terminal parameters are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted. This form should be used when changing parameters that will affect output.

TCSETXF The argument is a pointer to a termiox structure. The current terminal parameters are set from the values stored in that structure. The change occurs after all characters queued for output have been transmitted; all characters queued for input are discarded and then the change occurs.

FILES

Files in or under /dev/*.

SEE ALSO

ioctl(BA_OS), stty(AU_CMD), termio(BA_DEV).

LEVEL

Level 1.

NAME

ticlts, ticots, ticotsord – loopback transport providers

SYNOPSIS

```
#include <ticlts.h>
#include <ticots.h>
#include <ticotsord.h>
```

DESCRIPTION

The devices known as **ticlts**, **ticots**, and **ticotsord** are ''loopback transport providers,'' that is, stand-alone networks at the transport level. Loopback transport providers are transport providers in every sense except one: only one host (the local machine) is ''connected to'' a loopback network. Loopback transports present a TPI (**STREAMS**-level) interface to application processes and are intended to be accessed via the TLI (application-level) interface. They are implemented as clone devices and support address spaces consisting of ''flex-addresses,'' that is, arbitrary sequences of octets, of length > 0, represented by a **netbuf** structure.

ticlts is a datagram-mode transport provider. It offers (connectionless) service of type **T_CLTS**. Its default address size is **TCL_DEFAULTADDRSZ**. **ticlts** prints the following error messages [see **t_rcvuderr**(BA_LIB)]:

TCL_BADADDR	bad address specification
TCL_BADOPT	bad option specification
TCL_NOPEER	bound
TCL_PEERBADSTATE	peer in wrong state

ticots is a virtual circuit-mode transport provider. It offers (connection-oriented) service of type **T_COTS**. Its default address size is **TCO_DEFAULTADDRSZ**. **ticots** prints the following disconnect messages [see **t_rcvdis**(BA_LIB)]:

TCO_NOPEER	no listener on destination address
TCO_PEERNOROOMONQ	peer has no room on connect queue
TCO_PEERBADSTATE	peer in wrong state
TCO_PEERINITIATED	peer-initiated disconnect
TCO_PROVIDERINITIATED	provider-initiated disconnect

ticotsord is a virtual circuit-mode transport provider, offering service of type **T_COTS_ORD** (connection-oriented service with orderly release). Its default address size is **TCOO_DEFAULTADDRSZ**. **ticotsord** prints the following disconnect messages [see **t_rcvdis**(BA_LIB)]:

TCOO_NOPEER	no listener on destination address
TCOO_PEERNOROOMONQ	peer has no room on connect queue
TCOO_PEERBADSTATE	peer in wrong state
TCOO_PEERINITIATED	peer-initiated disconnect
TCOO_PROVIDERINITIATED	provider-initiated disconnect

USAGE

Loopback transports support a local IPC mechanism through the TLI interface. Applications implemented in a transport provider-independent manner on a client-server model using this IPC are transparently transportable to networked environments.

Transport provider-independent applications must not include the header files listed in the synopsis section above. In particular, the options are (like all transport provider options) provider dependent.

ticlts and **ticots** support the same service types (**T_CLTS** and **T_COTS**) supported by the OSI transport-level model. The use of **ticlts** and **ticots is encouraged.**

ticotsord supports the same service type (**T_COTS_ORD**) supported by the TCP/IP model. The use of **ticotsord** is discouraged except for reasons of compatibility.

FILES

/dev/ticlts
/dev/ticots
/dev/ticotsord

LEVEL

Level 1.

NAME

timod – Transport Interface cooperating STREAMS module

DESCRIPTION

timod is a STREAMS module for use with the Transport Interface (TI) functions of the Network Services library. The **timod** module converts a set of **ioctl**(BA_OS) calls into STREAMS messages that may be consumed by a transport protocol provider which supports the Transport Interface. This allows a user to initiate certain TI functions as atomic operations.

The **timod** module must be pushed onto only a stream terminated by a transport protocol provider which supports the TI.

All STREAMS messages, with the exception of the message types generated from the **ioctl** commands described below, will be transparently passed to the neighboring STREAMS module or driver. The messages generated from the following **ioctl** commands are recognized and processed by the **timod** module. The format of the **ioctl** call is:

```
#include <sys/stropts.h>
          -
          -
struct strioctl strioctl;
          -
          -
strioctl.ic_cmd = cmd;
strioctl.ic_timeout = INFTIM;
strioctl.ic_len = size;
strioctl.ic_dp = (char *)buf
ioctl(fildes, I_STR, &strioctl);
```

Where, on issuance, *size* is the size of the appropriate TI message to be sent to the transport provider and on return *size* is the size of the appropriate TI message from the transport provider in response to the issued TI message. *buf* is a pointer to a buffer large enough to hold the contents of the appropriate TI messages. The TI message types are defined in **sys/tihdr.h**. The possible values for the *cmd* field are:

TI_BIND Bind an address to the underlying transport protocol provider. The message issued to the **TI_BIND ioctl** is equivalent to the TI message type **T_BIND_REQ** and the message returned by the successful completion of the **ioctl** is equivalent to the TI message type **T_BIND_ACK**.

TI_UNBIND Unbind an address from the underlying transport protocol provider. The message issued to the **TI_UNBIND ioctl** is equivalent to the TI message type **T_UNBIND_REQ** and the message returned by the successful completion of the **ioctl** is equivalent to the TI message type **T_OK_ACK**.

TI_GETINFO Get the TI protocol specific information from the transport protocol provider. The message issued to the **TI_GETINFO ioctl** is equivalent to the TI message type **T_INFO_REQ** and the message

returned by the successful completion of the **ioctl** is equivalent to the TI message type **T_INFO_ACK**.

TI_OPTMGMT Get, set or negotiate protocol specific options with the transport protocol provider. The message issued to the **TI_OPTMGMT ioctl** is equivalent to the TI message type **T_OPTMGMT_REQ** and the message returned by the successful completion of the **ioctl** is equivalent to the TI message type **T_OPTMGMT_ACK**.

FILES

sys/timod.h
sys/tiuser.h
sys/tihdr.h
sys/errno.h

SEE ALSO

tirdwr(BA_DEV)

RETURN VALUE

If the **ioctl** system call returns with a value greater than 0, the lower 8 bits of the return value will be one of the TI error codes as defined in **sys/tiuser.h**. If the TI error is of type **TSYSERR**, then the next 8 bits of the return value will contain an error as defined in **sys/errno.h** [see **errno**(BA_ENV)].

LEVEL

Level 1.

NAME

tirdwr – Transport Interface read / write interface STREAMS module

DESCRIPTION

tirdwr is a STREAMS module that provides an alternate interface to a transport provider which supports the Transport Interface (TI) functions of the Network Services library (see Section BA_LIB). This alternate interface allows a user to communicate with the transport protocol provider using the **read**(BA_OS) and **write**(BA_OS) system calls. The **putmsg**(BA_OS) and **getmsg**(BA_OS) system calls may also be used. However, **putmsg** and **getmsg** can only transfer data messages between user and stream.

The **tirdwr** module must only be pushed [see **I_PUSH** in **streamio**(BA_DEV)] onto a stream terminated by a transport protocol provider which supports the TI. After the **tirdwr** module has been pushed onto a stream, none of the Transport Interface functions can be used. Subsequent calls to TI functions will cause an error on the stream. Once the error is detected, subsequent system calls on the stream will return an error with **errno** set to **EPROTO**.

The following are the actions taken by the **tirdwr** module when pushed on the stream, popped [see **I_POP** in **streamio**(BA_DEV)] off the stream, or when data passes through it.

push When the module is pushed onto a stream, it will check any existing data destined for the user to ensure that only regular data messages are present. It will ignore any messages on the stream that relate to process management, such as messages that generate signals to the user processes associated with the stream. If any other messages are present, the **I_PUSH** will return an error with **errno** set to **EPROTO**.

write The module will take the following actions on data that originated from a **write** system call:

All messages with the exception of messages that contain control portions (see the **putmsg** and **getmsg** system calls) will be trans-

Messages that represent expedited data will generate an error. All further system calls associated with the stream will fail with **errno** set to **EPROTO**.

Any data messages with control portions will have the control portions removed from the message prior to passing the message on to the upstream neighbor.

Messages that represent an orderly release indication from the transport provider will generate a zero length data message, indicating the end of file, which will be sent to the reader of the stream. The orderly release message itself will be freed by the module.

Messages that represent an abortive disconnect indication from the transport provider will cause all further **write** and **putmsg** system calls to fail with **errno** set to **ENXIO**. All further **read** and **getmsg** system calls will return zero length data (indicating end of file) once all previous data has been read.

With the exception of the above rules, all other messages with control portions will generate an error and all further system calls associated with the stream will fail with **errno** set to **EPROTO**.

Any zero length data messages will be freed by the module and they will not be passed onto the module's upstream neighbor.

pop When the module is popped off the stream or the stream is closed, the module will take the following action:

If an orderly release indication has been previously received, then an orderly release request will be sent to the remote side of the transport connection.

SEE ALSO

streams(BA_DEV), **timod**(BA_DEV)
getmsg(BA_OS), **putmsg**(BA_OS), **read**(BA_OS), **write**(BA_OS)

LEVEL

Level 1.

Kernel Extension Introduction

While the Base System is intended to support a run-time environment for executable applications, the Kernel Extension provides additional operating system services that will not be required by many application-programs but which are needed for some environments.

The Kernel Extension provides operating system services to support memory management facilities, process accounting tools, software development tools, and applications or tools that require more sophisticated inter-process communication than is provided by the Base System.

The Base System is prerequisite for support of the Kernel Extension.

SUMMARY OF OS SERVICE ROUTINES

The following OS service routines are supported by the Kernel Extension (exception: items marked with a sharp (#) are optional, hardware-dependent routines and will only appear on machines with the requisite hardware.) Items marked with a (†) are new to this extension. Items marked with a star (*) are Level 2, as defined in the *General Introduction* to this volume.

acct	modload†	msgget	plock*	semop
chroot	modpath†	msgrcv	priocntl†	shmat#
getksym†	modstat†	msgsnd	profil	shmctl#
mmap	moduload†	msync	ptrace	shmdt#
modadm†	mprotect	munmap	semctl	shmget#
modadmin†	msgctl	nice	semget	

priocntl has been added to this extension as the preferred interface for scheduling. It has been removed from the RT_OS extension.

The following routines have been added to this extension in support of Dynamically Loadable Kernel Modules: getkeysym, modpath, modadm, modstat, modadmin, moduload, modload. Dynamic installation of filesystem types, exec() modules, drivers, Streams modules and multiplexors will be supported. This feature provides the ability to add software to a running system in multi-user mode, without halting or or rebooting the system. [See Also modadmin(AS_CMD)]

Organization of Technical Information

The *Kernel Extensions Definitions* chapter defines terms used in manual page descriptions in later chapters.

The *Kernel Extension Environment* chapter describes elements of the assumed operating environment for this extension, including additional behavior of Base System components when the Kernel Extension is present on the system.

The *Kernel Extension OS Service Routines* chapter provides manual page descriptions of library routines supported by this extension.

Kernel Extension Environment Routines

The following section contains the manual pages for the KE_ENV routines.

FINAL COPY
June 15, 1995
File: ke_env.cov
svid

NAME

effects – effects of the Kernel Extension on the Base System

DESCRIPTION

Some of the Base System V operating system services are affected by the additional services in this extension. The effects are listed below for each routine:

exec(BA_OS)

The AFORK flag in the `ac_flag` field of the accounting record is turned off, and the `ac_comm` field is reset by executing an exec routine [see acct(KE_OS)].

Any process, data, or text-locks are removed and not inherited by the new process [see plock(KE_OS)].

Profiling is disabled for the new process [see profil(KE_OS)].

The shared-memory-segments attached to the calling process will not be attached to the new process [see shmop(KE_OS)].

The new process also inherits the following additional attributes from the calling process:

`nice` value [see nice(KE_OS)];

`semadj` values [see semop(KE_OS)];

exit(BA_OS)

An accounting record is written on the accounting file if the system's accounting routine is enabled [see acct(KE_OS)].

If the process has a process-lock, text-lock, or data-lock, the lock is removed [see plock(KE_OS)].

Each attached shared-memory-segment is detached and the value of `shm_nattch` in the data structure associated with its shared-memory-identifier is decremented by `1`.

For each semaphore for which the calling process has set a `semadj` value [see semop(KE_OS)], that `semadj` value is added to the `semval` of the specified semaphore.

fork(BA_OS)

The AFORK flag is turned on when the function `fork()` is executed.

The child process inherits the following additional attributes from the parent process:

The `ac_comm` contents of the accounting record [see acct(KE_OS)];

`nice` value [see nice(KE_OS)], scheduling priority and time quantum;

profiling on / off status [see profil(KE_OS)];

all attached shared-memory-segments [see shmop(KE_OS)].

The child process differs from the parent process in the following additional ways:

All `semadj` values are cleared [see semop(KE_OS)].

Process-locks, text-locks, and data-locks are not inherited by the child process [see plock(KE_OS), mctl(KE_OS), memctl(KE_OS), mlock(KE_OS), and mlockall(KE_OS)].

SEE ALSO

acct(KE_OS), chroot(BA_OS) mctl(KE_OS), memctl(KE_OS), mlock(KE_OS), mlockall(KE_OS), nice(KE_OS), plock(KE_OS), profil(KE_OS), semop(KE_OS), shmop(KE_OS)

LEVEL

Level 1.

NAME

error – error codes and condition definitions

SYNOPSIS

```
#include <errno.h>

‡ extern int errno;

errno
```

DESCRIPTION

The numerical value represented by the symbolic name of an error condition is assigned to `errno` for errors that occur when executing a system service routine or general library routine.

To be consistent with the C Standard, the interface definition of `errno` has been change in the SIVD, Fourth Edition. Programs should obtain the value of `errno` by including `<errno.h>`.

The macro `errno` expands to a modifiable *lvalue* that has type `int`, the value of which is set to a positive error number by several library functions. `errno` need not be the identifier of an object, *e.g.*, it might expand to a modifiable *lvalue* resulting from a function call. It is unspecified whether `errno` is a macro or an identifier declared with external linkage. If an `errno` macro definition is suppressed to access an actual object, or if a program defines an identifier with the name `errno`, the behavior is undefined.

In addition to the values defined in the Base System for the external variable `errno` [see errno(BA_ENV)], two additional error conditions are defined in the Kernel Extension:

`ENOMSG` No message of desired type.

An attempt was made to receive a message of a type that does not exist on the specified message queue [see msgop(KE_OS)].

`EIDRM` Identifier removed.

This error is returned to processes that resume execution because of the removal of an identifier [see msgctl(KE_OS), semctl(KE_OS), and shmctl(KE_OS)].

SEE ALSO

errno(BA_ENV), msgctl(KE_OS), msgop(KE_OS), semctl(KE_OS), shmctl(KE_OS).

LEVEL

Level 1.

NAME

sys/ipc.h – inter-process communication access structure

SYNOPSIS

```
#include <sys/ipc.h>
```

DESCRIPTION

The <sys/ipc.h> header uses three mechanisms for inter-process communication (IPC): messages, semaphores and shared memory. All use a common structure type, ipc_perm to pass information used in determining permission to perform an IPC operation.

The structure ipc_perm contains the following members:

```
uid_t    uid;      /* owner's user ID */
gid_t    gid;      /* owner's group ID */
uid_t    cuid;          /* creator's user ID */
gid_t    cgid;          /* creator's group ID */
mode_t   mode;          /* read/write permission */
```

Definitions are given for the following constants:

Mode bits:

IPC_CREAT	create entry if key doesn't exist
IPC_EXCL	fail if key exists
IPC_NOWAIT	error if request must wait

Keys:

IPC_PRIVATE	private key

Control Commands:

IPC_RMID	remove identifier
IPC_SET	set options
IPC_STAT	get options

LEVEL

Level 1.

NAME

sys/msg.h – message queue structures

SYNOPSIS

```
#include <sys/msg.h>
```

DESCRIPTION

The <sys/msg.h> header defines the following constant and members of the structure msqid_ds

Message operation flag:

MSG_NOERROR no error if big message

The structure msqid_ds contains the following members:

```
struct ipc_perm  msg_perm;   /* operation permission
                                structure */
unsigned long    msg_qnum;   /* number of messages
                                currently on queue */
unsigned long    msg_qbytes; /* max number of bytes
                                allowed on queue */
pid_t            msg_lspid;  /* pid of last msgsnd() */
pid_t            msg_lrpid;  /* pid of last msgrcv() */
time_t           msg_stime;  /* time of last msgsnd() */
time_t           msg_rtime;  /* time of last msgrcv() */
time_t           msg_ctime;  /* time of last change */
```

msg_perm is an ipc_perm structure [see ipc(KE_ENV)] that specifies the message operation permission.

msg_qnum is the number of messages currently on the queue.

msg_qbytes is the maximum number of bytes allowed on the queue.

msg_lspid is the process ID of the last process that performed a msgsnd operation.

msg_lrpid is the process ID of the last process that performed a msgrcv operation.

msg_stime is the time of the last msgsnd operation.

msg_rtime is the time of the last msgrcv operation.

msg_ctime is the time of the last msgctl operation that changed a member of the above structure.

The following are declared as either functions or macros:

```
msgctl()     msgrcv()
msgget()     msgsnd()
```

SEE ALSO

ipc(KE_ENV), msgctl(KE_OS), msgget(KE_OS), msgop(KE_OS).

LEVEL

Level 1.

NAME

sys/sem.h – semaphore facility

SYNOPSIS

```
#include <sys/sem.h>
```

DESCRIPTION

The <sys/sem.h> header defines the following constants and structures.

Semaphore operation flags:

SEM_UNDO	set up adjust on exit entry

Command definitions for the function semctl() [see semctl(KE_OS)]:

GETNCNT	get semncnt
GETPID	get sempid
GETVAL	get semval
GETALL	get all semvals
GETZCNT	get semzcnt
SETVAL	set semval
SETALL	set all semvals

The structure semid_ds contains the following members:

```
struct ipc_perm  sem_perm;   /* operation permission
                                structure */
ushort           sem_nsems;  /* number of semaphores
                                in set */
time_t           sem_otime;  /* last semop() time */
time_t           sem_ctime;  /* last time changed by
                                semctl() */
```

sem_perm is an ipc_perm structure that specifies the semaphore operation permission [see ipc(KE_ENV)].

sem_nsems is a value that is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a non-negative integer referred to as a sem_num. The value of sem_num runs sequentially from 0 to the value of sem_nsems–1. sem_otime is the time of the last semop operation, and sem_ctime is the time of the last semctl operation that changed a member of the above structure.

semval is a non-negative integer.

sempid is equal to the process ID of the last process that performed a semaphore operation on this semaphore.

semncnt is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value.

semzcnt is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

The number of semaphores in a set is `sem_nsems` within the set semaphores number from 0 to `sem_nsems`-1. The number of a semaphore is known as a `sem_num`.

A semaphore is represented by an anonymous structure containing the following members:

```
ushort    semval;   /* semaphore value */
pid_t     sempid;   /* pid of last operation */
ushort    semncnt;  /* number of processes waiting
                       for semval to become greater
                       than current value */
ushort    semzcnt;  /* number of processes waiting
                       for semval to become zero */
```

The structure `sembuf` contains the following members:

```
ushort    sem_num;   /* semaphore number */
short     sem_op;    /* semaphore operation */
short     sem_flg;   /* operation flags */
```

The following are declared as either functions or macros:

```
semctl()   semget()   semop()
```

SEE ALSO

ipc(KE_OS), semctl(KE_OS), semget(KE_OS), semop(KE_OS).

LEVEL

Level 1.

NAME

sys/shm.h – shared memory facility

SYNOPSIS

```
#include <sys/shm.h>
```

DESCRIPTION

The <sys/shm.h> header defines the following constants and the structure.

Message operation flags:

SHM_RDONLY	attach read-only (else read-write)
SHMLBA	segment low boundary address multiple
SHM_RND	round attach address to SHMLBA

The structure shmid_ds contains the following members:

```
struct ipc_perm  shm_perm;   /* operation permission
                                structure */
int              shm_segsz;  /* segment size in bytes */
pid_t            shm_lpid;   /* pid of last shmop */
pid_t            shm_cpid;   /* pid of creator */
unsigned long    shm_nattch; /* number of current
                                attaches */
time_t           shm_atime;  /* time of last shmat() */
time_t           shm_dtime;  /* time of last shmdt() */
time_t           shm_ctime;  /* time of last change by
                                shmctl() */
```

shm_perm is an ipc_perm structure that specifies the shared memory operation permission [see ipc(KE_ENV)].

shm_segsz specifies the size of the shared memory segment.

shm_cpid is the process ID of the process that created the shared memory identifier.

shm_lpid is the process ID of the last process that performed a shmop() routine [see shmop(KE_OS)].

shm_nattch is the number of processes that currently have this segment attached.

shm_atime is the time of the last shmat operation.

shm_dtime is the time of the last shmdt operation. is the time of the last shmctl operation that changed one of the members of the above structure.

The following are declared as either functions or macros:

shmat() shmctl() shmdt() shmget()

SEE ALSO

ipc(KE_ENV), shmctl(KE_OS), shmget(KE_OS), shmop(KE_OS).

LEVEL

Level 1.

Kernel Extension OS Service Routines

The following section contains the manual pages for the KE_OS routines.

FINAL COPY
June 15, 1995
File: ke_os.cov
svid

NAME

acct – enable or disable process accounting

SYNOPSIS

```
#include <unistd.h>

int acct(const char *path);
```

DESCRIPTION

acct enables or disables the system process accounting routine. If the routine is enabled, an accounting record will be written in an accounting file for each process that terminates. The termination of a process can be caused by one of two things: an **exit** call or a signal The calling process must have the appropriate privilege (**P_SYSOPS**) to enable or disable accounting.

path points to a pathname naming the accounting file. An accounting file produced as a result of calling the **acct** function has records in the format defined by the structure **acct** in **<sys/acct.h>**, which defines the following data type:

```
comp_t  /* floating point - 13-bit fraction, */
        /*                  3-bit exponent  */
```

The structure **acct** includes the following members:

```
char   ac_flag;     /* Accounting flag */
char   ac_stat;     /* Exit status */
uid_t  ac_uid;      /* Accounting user ID */
gid_t  ac_gid;      /* Accounting group ID */
dev_t  ac_tty;      /* controlling tty */
time_t ac_btime;    /* Beginning time */
comp_t ac_utime;    /* accounting user time in clock ticks */
comp_t ac_stime;    /* accounting system time in clock ticks */
comp_t ac_etime;    /* accounting elapsed time in clock ticks */
comp_t ac_mem;      /* memory usage in clicks */
comp_t ac_io;       /* chars transferred by read/write */
comp_t ac_rw;       /* number of block reads/writes */
char   ac_comm[8];  /* command name */
```

and defines the following symbolic names:

```
AFORK  /* has executed fork, but no exec */
ASU    /* used appropriate privileges */
ACCTF  /* record type: 00 = acct */
```

The **ac_stat** value is the status returned in the argument to **wait** [see **wait**(BA_OS)] cast to a **char**.

The **AFORK** flag is set in **ac_flag** when the **fork** routine is executed and reset when an **exec** routine is executed [see **exec**(BA_OS)]. The **ac_comm** field is inherited from the parent process when a child process is created with the **fork** routine and is reset when an exec routine is executed. The variable **ac_mem** is a cumulative record of memory usage and is incremented each time the system charges the process with a clock tick.

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is **(char *)NULL** and no errors occur during the system call.

Return Values

On success, **acct** returns 0. On failure, **acct** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **acct** fails and sets **errno** to:

EACCES	The file named by *path* is not an ordinary file.
EACCES	Search permission is denied on a component of the path prefix.
EACCES	Write permission on the name file is denied.
EFAULT	*path* points to an illegal address.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while **_POSIX_NO_TRUNC** is in effect.
ENOTDIR	A component of the path prefix is not a directory.
ENOENT	One or more components of the accounting file pathname do not exist.
EPERM	The calling process does not have the appropriate privilege to enable or disable accounting.
EROFS	The named file resides on a read-only file system.

SEE ALSO

exit(BA_OS)

LEVEL

Level 1.

NAME

chroot — change root directory

SYNOPSIS

```
int chroot(const char *path);
```

DESCRIPTION

The function `chroot()` causes the named directory to become the root directory, the starting point for *path* searches for absolute pathnames. The function `chroot()` does not affect the user's working directory.

The argument *path* points to a pathname naming a directory.

The process must have appropriate privileges to change the root directory.

The `..` entry in the root directory is interpreted to mean the root directory itself. Thus, `..` cannot be used to access files outside the sub-tree rooted in the root directory.

RETURN VALUE

Upon successful completion, the function `chroot()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error. On failure the root directory remains unchanged.

ERRORS

Under the following conditions, the function `chroot()` fails, and sets `errno` to:

`EACCES` if search permission is denied for a component of *path*.

`ENOTDIR` if any component of the pathname is not a directory.

`ENOENT` if the named directory does not exist or *path* points to an empty string.

`EPERM` if the process does not have appropriate privileges.

`ENAMETOOLONG`
if the size of a pathname exceeds `{PATH_MAX}`, or a pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

`ELOOP` if too many symbolic links are encountered in translating the path.

SEE ALSO

chdir(BA_OS).

LEVEL

Level 1.

NAME

exit, **_exit** – terminate process

SYNOPSIS

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

DESCRIPTION

_exit terminates the calling process with the following consequences:

All of the file descriptors, directory streams and message catalogue descriptors open in the calling process are closed.

A **SIGCHLD** signal is sent to the calling process's parent process.

If the parent process of the calling process has not specified the **SA_NOCLDWAIT** flag [see **sigaction**(BA_OS)], the calling process is transformed into a ''zombie process.'' A zombie process is a process that only occupies an entry in the process list. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information [see **<sys/proc.h>**] to be used by the **times** system call.

The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a **semadj** value [see **semop**(KE_OS)], that **semadj** value is added to the **semval** of the specified semaphore.

If the process has a process, text, or data lock, an **unlock** is performed [see **plock**(KE_OS)].

An accounting record is written on the accounting file if the system's accounting routine is enabled [see **acct**(AS_CMD)].

If the process is a controlling process, **SIGHUP** is sent to the foreground process group of its controlling terminal and its controlling terminal is deallocated.

If the calling process has any stopped children whose process group will be orphaned when the calling process exits, or if the calling process is a member of a process group that will be orphaned when the calling process exits, that process group will be sent **SIGHUP** and **SIGCONT** signals.

The C function **exit** calls any functions registered through the **atexit** function in the reverse order of their registration. The function **_exit** circumvents all such functions and cleanup.

The symbols **EXIT_SUCCESS** and **EXIT_FAILURE** are defined in **stdlib.h** and may be used as the value of *status* to indicate successful or unsuccessful termination, respectively.

SEE ALSO

acct(AS_CMD), **plock**(KE_OS), **semop**(KE_OS), **sigaction**(BA_OS), **times**(BA_OS), **wait**(BA_OS).

LEVEL

Level 1.

NAME

getksym – get information for a global kernel symbol

SYNOPSIS

```
#include <sys/ksym.h>

int getksym(char *symname, unsigned long *value, unsigned long *info);
```

DESCRIPTION

getksym, given a *symname*, looks for a global symbol of that name in the symbol table of the running kernel (including all currently loaded kernel modules). If it finds a match, **getksym** returns the value associated with that symbol (typically its address) in the space pointed to by *value*, and the type of that symbol in the space pointed to by *info*. If more than one symbol of the given name exists in the search space, the one (if any) in the statically bound kernel or, if not there, the first one found among the loaded modules will be returned.

If **getksym** is given a valid address in the running kernel in the space pointed to by *value*, it will return, in the space pointed to by *symname*, the name of the symbol whose value is the closest one less than or equal to the given value and, in space pointed to by *info*, the difference between the address given and the value of the symbol found.

Return Values

On failure, **getksym** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **getksym** fails and sets **errno** to:

EFAULT Invalid pointer for *symname*, *value*, or *info*

ENAMETOOLONG The length of the symbol name exceeds the maximum length of the characters.

ENOMATCH *symname* is not found in the running kernel (including loaded modules) or *value* is outside the range of the static kernel and any loaded modules.

SEE ALSO

nlist(SD_LIB),

LEVEL

Level 1.

NOTICES

As a consequence of the dynamically loadable kernel modules feature, a dynamic symbol table is now kept in the kernel address space representing all defined global symbols in the static kernel and all currently loaded modules. When a module is loaded, its symbol information is added to this table; when a module is unloaded, its symbol information is deleted.

Finding out the address of a particular kernel variable was commonly done by using **nlist**(SD_LIB) on **/stand/unix**. This is no longer an accurate way to get that information, since **/stand/unix** only contains the symbol table for the static kernel. The symbol tables for the loadable modules are elsewhere on the system, but which modules are loaded and from where changes over time. So, as part of this feature, two new ways of getting at information associated with kernel symbols

have been provided.

The **getksym**(KE_OS) system call provides the kind of information on a given kernel symbol or address that **nlist**(SD_LIB) provided. However, the symbol name/address association may not be valid by the time it is returned to the user (for example, if the symbol is defined in a loadable module and that module is unloaded), unless the user takes special steps like keeping the module loaded by making sure there is an outstanding **open**, **mount**, . . .

Because of this later complication and because most interest in kernel addresses is related to reading or writing from **/dev/kmem**, an alternate atomic method of reading and writing in the kernel address space based on a symbol name is provided. Three new ioctl commands now exist in the **mm** memory driver for the **/dev/kmem** minor device In this way, a user gets the desired IO operation accomplished without fear that a module may be unloaded in the middle. Of course, this user must still open **/dev/kmem** for the correct type of IO and so the appropriate protections against unauthorized access still exist.

NAME

mmap – map pages of memory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

caddr_t mmap(caddr_t addr, size_t len, int prot,
     int flags, int fd, off_t off);
```

DESCRIPTION

The function `mmap()` establishes a mapping between a process's address space and a virtual memory object. The format of the call is as follows:

pa=`mmap(`*addr*`,` *len*`,` *prot*`,` *flags*`,` *fd*`,` *off*`);`

`mmap()` establishes a mapping between the process's address space at an address *pa* for *len* bytes to the memory object represented by the file descriptor *fd* at offset *off* for *len* bytes. The value of *pa* is an implementation-dependent function of the parameter *addr* and values of *flags*, further described below. A successful `mmap()` call returns *pa* as its result. The address ranges covered by [*pa, pa + len*) and [*off, off + len*) must be legitimate for the possible (not necessarily current) address space of a process and the object in question, respectively.

The mapping established by `mmap()` replaces any previous mappings for the process's pages in the range [*pa, pa + len*).

The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the pages being mapped. The protection options are defined in `<sys/mman.h>` as:

```
PROT_READ           /* page can be read */
PROT_WRITE          /* page can be written */
PROT_EXEC           /* page can be executed */
PROT_NONE           /* page can not be accessed */
```

Not all implementations literally provide all possible combinations. `PROT_WRITE` is often implemented as `PROT_READ|PROT_WRITE` and `PROT_EXEC` as `PROT_READ|PROT_EXEC`. However, no implementation will permit a write to succeed where `PROT_WRITE` has not been set. The behavior of `PROT_WRITE` can be influenced by setting `MAP_PRIVATE` in the *flags* parameter, described below.

The parameter *flags* provides other information about the handling of the mapped pages. The options are defined in `<sys/mman.h>` as:

```
MAP_SHARED          /* Share changes */
MAP_PRIVATE         /* Changes are private */
MAP_FIXED           /* Interpret addr exactly */
```

`MAP_SHARED` and `MAP_PRIVATE` describe the disposition of write references to the memory object. If `MAP_SHARED` is specified, write references will change the memory object. If `MAP_PRIVATE` is specified, the initial write reference will create a private copy of the memory object page and redirect the mapping to the copy. Either `MAP_SHARED` or `MAP_PRIVATE` must be specified, but not both. The mapping type is retained across a `fork()`.

Note that the private copy is not created until the first write; until then, other users who have the object mapped MAP_SHARED can change the object.

MAP_FIXED informs the system that the value of *pa* must be *addr*, exactly. The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of system resources.

When MAP_FIXED is not set, the system uses *addr* in an implementation-defined manner to arrive at *pa*. The *pa* so chosen will be an area of the address space which the system deems suitable for a mapping of *len* bytes to the specified object. All implementations interpret an *addr* value of zero as granting the system complete freedom in selecting *pa*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a value for *pa*, it will never place a mapping at address 0, nor will it replace any extant mapping, nor map into areas considered part of the potential data or stack segments .

The parameter *off* is constrained to be aligned and sized according to the value returned by sysconf(). When MAP_FIXED is specified, the parameter *addr* must also meet these constraints. The system performs mapping operations over whole pages. Thus, while the parameter *len* need not meet a size or alignment constraint, the system will include, in any mapping operation, any partial page specified by the range [*pa, pa + len*).

The system will always zero-fill any partial page at the end of an object. Further, the system will never write out any modified portions of the last page of an object which are beyond its end. References to whole pages following the end of an object will result in the delivery of a SIGBUS signal. SIGBUS signals may also be delivered on various file system conditions, including quota exceeded errors.

mmap() adds an extra reference to the object associated with the file descriptor *fd* which is not removed by a subsequent close() on that file descriptor. This reference is removed when the entire range is unmapped (explicitly or implicitly).

RETURN VALUE

Upon successful completion, the function mmap() returns the address at which the mapping was placed (*pa*); otherwise, it returns a value of –1 and sets errno to indicate an error.

ERRORS

Under the following conditions, the function mmap() fails and sets errno to:

EAGAIN if the mapping could not be locked in memory.

EBADF if *fd* is not open.

EACCES if *fd* is not open for read, regardless of the protection specified, or *fd* is not open for write and PROT_WRITE was specified for a MAP_SHARED type mapping.

ENXIO if addresses in the range [*off, off + len*) are invalid for *fd*.

EINVAL if the arguments *addr* (if MAP_FIXED was specified) or *off* are not multiples of the page size as returned by sysconf().

EINVAL	if the field in *flags* is invalid (neither MAP_PRIVATE or MAP_SHARED).
ENODEV	if *fd* refers to an object for which mmap() is meaningless, such as a terminal.
ENOMEM	if MAP_FIXED was specified, and the range [*addr*, *addr* + *len*) exceeds that allowed for the address space of a process; or if MAP_FIXED was not specified and there is insufficient room in the address space to effect the mapping.

USAGE

The function mmap() allows access to resources via address space manipulations, instead of the read()/write() interface. Once a file is mapped, all a process has to do to access it is use the data at the address to which the object was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use mmap(),

```
fd = open(...)
lseek(fd, some_offset)
read(fd, buf, len)
/* use data in buf */
```

becomes

```
fd = open(...)
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fd, some_offset)
/* use data at address */
```

SEE ALSO

fcntl(BA_OS), fork(BA_OS), lockf(BA_OS), mlockall(RT_OS), munmap(KE_OS), mprotect(KE_OS), plock(KE_OS), sysconf(BA_OS).

LEVEL

Level 1.

NAME

modload – load a loadable kernel module on demand

SYNOPSIS

```
#include <sys/mod.h>

int modload(const char *pathname);
```

DESCRIPTION

modload allows processes with the appropriate privilege to demand-load a loadable module into a running system.

pathname gives the pathname of the module to be loaded, specified either as a module name or as an absolute pathname. If *pathname* specifies a module name, **modload** searches for the module's object file on disk in the list of directories set by **modpath**(KE_OS) (including the default directory **/etc/conf/mod.d**). If *pathname* specifies an absolute pathname, only *pathname* is used to locate the module's object file.

Tasks performed during the load operation include:

open the module's object file on disk

allocate kernel memory to hold the module

read the module's object file into memory

load any modules upon which the module depends that are not already loaded

relocate the module's symbols

resolve any external references to kernel symbols made by the module

execute the module's wrapper routine to perform any setup the module requires to initialize itself

logically link the module to the running kernel by creating the module's switch table entries

set a flag that prevents the module from being unloaded by the kernel auto-unload mechanism

Return Values

On success, **modload** returns the integer module id of the loaded module. On failure, **modload** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **modload** fails and sets **errno** to:

EACCES Search permission was denied by a *pathname* component.

ENOENT The file *pathname* does not exist.

EINVAL The file *pathname* is not preconfigured for dynamic loading or has invalid dependencies on other modules (such as a circular dependency).

ERELOC Error occurred processing the module's object file, or the module references symbols not defined in the running kernel, or the module references symbols in another loadable module, but it did not define its dependence on this module in its **Master** file.

ENAMETOOLONG *pathname* is more than **MAXPATHLEN** characters long.

ENOSYS Unable to perform the requested operation because the loadable modules functions are not configured into the system.

SEE ALSO

modadmin(AS_CMD), **modpath**(KE_OS), **modstat**(KE_OS), **moduload**(KE_OS)

LEVEL

Level 1.

NAME

modpath – change loadable kernel modules search path

SYNOPSIS

```
#include <sys/mod.h>

int modpath(const char *pathname);
```

DESCRIPTION

modpath allows processes with the appropriate privilege to modify the global search path used to locate object files for loadable kernel modules on disk. The search path modifications take effect immediately and affect all subsequent loads and all users on the system. Affected loads include all auto-loads performed by the kernel auto-load mechanism and all demand-loads performed by **modload**(KE_OS) using a module name.

pathname can specify a colon-separated list of absolute pathnames, or an absolute pathname, or **NULL**.

If *pathname* specifies a pathname, the named directories:

will be searched prior to searching any directories specified by previous calls to **modpath**

will be searched prior to searching the default loadable modules search path, which is always searched and always searched last

do not have to exist on the system at the time **modpath** is called

do not have to exist on the system at the time the load takes place

If *pathname* is equal to **NULL**, the loadable modules search path is reset to its default value

Return Values

On success, **modpath** returns 0. On failure, **modpath** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **modpath** fails and sets **errno** to:

EINVAL	List of directories specified by *pathname* is malformed.
ENAMETOOLONG	*pathname* is more than **MAXPATHLEN** characters long.
ENOSYS	Unable to perform the requested operation because the loadable modules functions are not configured into the system.

SEE ALSO

modadmin(AS_CMD), **modload**(KE_OS)

LEVEL

Level 1.

NAME

modstat – get information for loadable kernel modules

SYNOPSIS

```
#include <sys/mod.h>

int modstat(int modid, struct modstatus *stbuf, boolean_t next_modid);
```

DESCRIPTION

modstat allows processes with the appropriate privilege to obtain information about the currently loaded loadable kernel modules. Any module that has been loaded by the kernel auto-load mechanism or demand-loaded by **modload**(KE_OS) may be queried by **modstat**.

When passed the module identifier *modid*, **modstat** fills up the members of the **modstatus** structure pointed to by *strbuf* with information about that module.

If the value of *next_modid* is **B_TRUE**, **modstat** fills up a **modstatus** structure with information about the module whose module identifier is greater than or equal to *modid*.

Return Values

On success, **modstat** returns one or more **modstatus** structures. On failure, **modstat** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **modstat** fails and sets **errno** to:

EINVAL *modid* does not match the identifier for any currently loaded module when *next_modid* is **B_FALSE** or *modid* is greater than the identifier for any currently loaded module when *next_modid* is **B_TRUE**.

ENOSYS Unable to perform the requested operation because the loadable modules functions are not configured into the system.

SEE ALSO

modadmin(AS_CMD), **modload**(KE_OS), **moduload**(KE_OS)

LEVEL

Level 1.

NAME

moduload – unload a loadable kernel module on demand

SYNOPSIS

```
#include <sys/mod.h>

int moduload(int modid);
```

DESCRIPTION

moduload allows processes with the appropriate privilege to demand-unload a loadable module—or all loadable modules—from a running system.

If *modid* specifies a module identifier, **moduload** attempts to unload that module. If *modid* specifies **0** (zero), **moduload** attempts to unload all loadable modules.

Loadable modules are considered unloadable if all of the following conditions are true:

- the module is not currently being used
- the module is not currently being loaded or unloaded
- no module that depends on the module is currently loaded
- profiling is disabled

When **moduload** finds that it cannot demand-unload a module for one of the reasons cited above, it flags the module as a candidate for subsequent unloading by the kernel's auto-unload mechanism.

Tasks performed during the unload operation include:

- logically disconnect the module from the running system by removing the module's switch table entry
- execute the module's wrapper routine to perform any cleanup the module requires to remove itself from the system
- free kernel memory allocated for the module

Return Values

On success, **moduload** returns 0. On failure, **moduload** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **moduload** fails and sets **errno** to:

EBUSY Outstanding references to this module exist, or modules that depend on this module are currently loaded, or profiling is not enabled, or this module is in the process of being loaded or unloaded.

EINVAL *modid* does not specify a valid loadable module identifier, or *modid* is not currently loaded.

ENOSYS Unable to perform the requested operation because the loadable modules functions are not configured into the system.

SEE ALSO

modadmin(AS_CMD), **modload**(KE_OS), **modpath**(KE_OS), **modstat**(KE_OS)

LEVEL

Level 1.

NAME

mprotect – set protection of memory mapping

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

mprotect(caddr_t addr, size_t len, int prot);
```

DESCRIPTION

The function `mprotect()` changes the access protections on the mappings specified by the range [*addr*, *addr* + *len*) to be that specified by *prot*. Legitimate values for *prot* are the same as those permitted for `mmap()` and are defined in `<sys/mman.h>` as:

```
PROT_READ           /* page can be read */
PROT_WRITE          /* page can be written */
PROT_EXEC           /* page can be executed */
PROT_NONE           /* page can not be accessed */
```

RETURN VALUE

Upon successful completion, the function `mprotect()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `mprotect()` fails and sets `errno` to:

`EACCES` if *prot* specifies a protection that violates the access permission the process has to the underlying memory object.

`EAGAIN` if *prot* specifies `PROT_WRITE` over a `MAP_PRIVATE` mapping and there are insufficient memory resources to reserve for locking the private page.

`EINVAL` if *addr* is not a multiple of the page size as returned by `sysconf()`.

`ENOMEM` if addresses in the range [*addr*, *addr* + *len*) are invalid for the address space of a process, or specify one or more pages which are not mapped.

When `mprotect()` fails for reasons other than `EINVAL`, the protections on some of the pages in the range [*addr*, *addr* + *len*) will have been changed. If the error occurs on some page at *addr2*, then the protections of all whole pages in the range [*addr*, *addr2*] will have been modified.

SEE ALSO

mmap(KE_OS), memcntl(RT_OS), mlock(RT_OS), mlockall(RT_OS), plock(KE_OS), sysconf(BA_OS).

LEVEL

Level 1.

NAME

msgctl – message control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, .../* struct msqid_ds *buf */);
```

DESCRIPTION

msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in the kernel extension definition examples.

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only access permission bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user **ID** equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*, or by a process that has the appropriate privileges. Only a user with appropriate privileges may raise the value of **msg_qbytes**.

IPC_RMID Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user **ID**

SEE ALSO

msgop(KE_OS)

LEVEL

Level 1.

NAME

msgget – get message queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

DESCRIPTION

msgget returns the message queue identifier associated with *key*. This identifier is accessible by any process in the system, subject to normal access restrictions and the permissions set with *msgflg*.

A successful call to **msgget** does not imply access to the queue in question, only a successful name mapping from *key* to ID.

A message queue identifier and associated message queue and data structure are created for *key* if one of the following are true:

key is **IPC_PRIVATE**.

key does not already have a message queue identifier associated with it, and (*msgflg***&IPC_CREAT**) is true.

On creation, the data structure associated with the new message queue identifier is initialized as follows:

msg_perm.cuid, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set to the effective user **ID** and effective group **ID**, respectively, of the calling process.

The low-order 9 bits of **msg_perm.mode** are set to the low-order 9 bits of *msgflg*.

msg_qnum, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set to 0.

msg_ctime is set to the current time.

msg_qbytes is set to the system limit.

Return Values

On success, **msgget** returns a non-negative integer, namely a message queue identifier. On failure, **msgget** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **msgget** fails and sets **errno** to:

EACCES	A message queue identifier exists for *key*, but the queue was not created supporting the specified operation permissions.
ENOENT	A message queue identifier does not exist for *key* and (*msgflg***&IPC_CREAT**) is false.
ENOSPC	A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

EEXIST	A message queue identifier exists for *key* but (*msgflg***&IPC_CREAT**) and (*msgflg***&IPC_EXCL**) are both true.

SEE ALSO

msgctl(KE_OS), **msgop**(KE_OS)

LEVEL

Level 1.

NAME

msgop: **msgsnd**, **msgrcv** – message operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp,
      size_t msgsz, int msgflg);

int msgrcv(int msqid, void *msgp,
      size_t msgsz, long msgtyp, int msgflg);
```

DESCRIPTION

msgsnd sends a message to the queue associated with the message queue identifier specified by *msqid*. *msgp* points to a user defined buffer that must contain first a field of type long integer that will specify the type of the message, and then a data portion that will hold the text of the message. The following is an example of members that might be in a user defined buffer.

```
long mtype;    /* message type */
char mtext[];  /* message text */
```

mtype is a positive integer that can be used by the receiving process for message selection. **mtext** is any text of length *msgsz* bytes. *msgsz* can range from **0** to a system imposed maximum.

msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes**

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg***&IPC_NOWAIT**) is true, the message is not sent and the caller returns immediately.

If (*msgflg***&IPC_NOWAIT**) is false, the caller suspends execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in

msg_qnum is incremented by 1.

msg_lspid ID of the caller.

msg_stime is set to the current time.

msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user defined structure pointed to by *msgp*. The structure must contain a message type field followed by the area for the message text (see the structure **mymsg** above). **mtype** is the received message's type as specified by the sending process. **mtext** is the text of the message. *msgsz* specifies the size in bytes of **mtext**. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg***&MSG_NOERROR**) is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp specifies the type of message requested as follows:

If *msgtyp* is 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg***&IPC_NOWAIT**) is true, the caller returns immediately with a return value of –1 and sets **errno** to **ENOMSG**.

If (*msgflg***&IPC_NOWAIT**) is false, the caller suspends execution until one of the following occurs:

A message of the desired type is placed on the queue.

msqid is removed from the system. When this occurs, **errno** is set to **EIDRM**, and a value of –1 is returned.

The caller receives a signal that is to be caught. In this case a message is not received and the caller resumes execution in the manner prescribed in **signal**(BA_OS).

On success, the following actions are taken with respect to the data structure associated with *msqid*

msg_qnum is decremented by 1.

msg_lrpid is set to the process **ID** of the caller.

msg_rtime is set to the current time.

Return Values

On success:

msgsnd returns 0.

msgrcv returns the number of bytes actually placed into *mtext*.

On failure, **msgsnd** and **msgrcv** return –1 and set **errno** to identify the error.

Errors

In the following conditions, **msgsnd** and **msgrcv** fail and set **errno** to:

EINTR **msgsnd** or **msgrcv** returned due to the receipt of a signal.

EIDRM **msgsnd** or **msgrcv** returned due to removal of *msqid* from the system.

In the following conditions, **msgsnd** fails and sets **errno** to:

EINVAL	*msqid* is not a valid message queue identifier.
EACCES	Operation permission is denied to the caller.
EINVAL	*mtype* is less than 1.
EAGAIN	The message cannot be sent for one of the reasons cited above and (*msgflg***&IPC_NOWAIT**) is true.
EINVAL	*msgsz* is less than zero or greater than the system-imposed limit.

In the following conditions, **msgrcv** fails and sets **errno** to:

EINVAL	*msqid* is not a valid message queue identifier.
EACCES	Operation permission is denied to the caller.
EINVAL	*msgsz* is less than 0.
E2BIG	The length of *mtext* is greater than *msgsz* and (*msgflg***&MSG_NOERROR**) is false.
ENOMSG	The queue does not contain a message of the desired type and (*msgtyp***&IPC_NOWAIT**) is true.

SEE ALSO

msgctl(KE_OS) **msgget**(KE_OS) **signal**(BA_OS)

LEVEL

Level 1.

NOTICES

Considerations for Threads Programming

While one thread is blocked, siblings might still be executing.

NAME

msync – synchronize memory with physical storage

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

int msync(caddr_t addr; size_t len; int flags);
```

DESCRIPTION

The function `msync()` writes all modified copies of pages over the range [*addr, addr + len*) to their permanent storage locations. `msync()` optionally invalidates any copies so that further references to the pages will be obtained by the system from their permanent storage locations.

flags is a bit pattern built from the following flags used to control the behavior of the operation:

`MS_ASYNC`	perform asynchronous writes
`MS_SYNC`	perform synchronous writes
`MS_INVALIDATE`	invalidate mappings

`MS_ASYNC` returns immediately once all write operations are scheduled; with `MS_SYNC` the system call will not return until all write operations are completed.

`MS_INVALIDATE` invalidates all cached copies of data in memory, so that further references to the pages will be obtained by the system from their permanent storage locations. This operation should be used by applications that require a memory object to be in a known state.

RETURN VALUE

Upon successful completion, the function `msync()` returns a value of `0`; otherwise, it returns a value of `–1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `msync()` fails and sets `errno` to:

`EBUSY` if some or all the addresses in the range [*addr, addr + len*) are locked.

`EINVAL` if *addr* is not a multiple of the page size as returned by `sysconf()`.

`ENOMEM` if some or all the addresses in the range [*addr, addr + len*) are invalid for the address space of the process or pages not mapped are specified.

USAGE

`msync()` should be used by programs that require a memory object to be in a known state, for example in building transaction facilities.

SEE ALSO

mmap(KE_OS), sysconf(BA_OS).

LEVEL

Level 1.

NAME

munmap – unmap pages of memory.

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

munmap(caddr_t addr, size_t len);
```

DESCRIPTION

The function `munmap()` removes the mappings for pages in the range [*addr*, *addr* + *len*). Further references to these pages will result in the delivery of a `SIGSEGV` signal to the process.

The function `mmap()` often performs an implicit `munmap()`.

RETURN VALUE

Upon successful completion, the function `munmap()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `munmap()` fails and sets `errno` to:

`EINVAL` if *addr* is not a multiple of the page size as returned by `sysconf()`.

`EINVAL` if addresses in the range [*addr*, *addr* + *len*) are outside the valid range for the address space of a process.

SEE ALSO

mmap(KE_OS), sysconf(BA_OS).

LEVEL

Level 1.

NAME

nice – change priority of a time-sharing process

SYNOPSIS

```
#include <unistd.h>

int nice(int incr);
```

DESCRIPTION

nice allows a member of the time-sharing scheduling class to change its priority.

nice adds the value of *incr* to the nice value of the calling process. The nice value is a non-negative number for which a more positive value results in lower CPU priority.

A maximum nice value of **NZERO** are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

Return Values

On success, **nice** returns the new nice value minus **NZERO**. On failure, **nice** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **nice** fails and sets **errno** to:

EPERM	**incr** is negative or greater than **NZERO** and the effective user ID of the calling process does not have the appropriate privilege.
EINVAL	The process was in a scheduling class other than time-sharing.

USAGE

priocntl(RT_CMD) is a more general interface to scheduler functions.

SEE ALSO

exec(BA_OS), **nice**(AS_CMD), **priocntl**(RT_CMD)

LEVEL

Level 1.

NAME

plock – lock into memory or unlock process, text, or data

SYNOPSIS

```
#include <sys/lock.h>

int plock(int op);
```

DESCRIPTION

plock allows the calling process to lock into memory or unlock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock). Locked segments are immune to all routine swapping. **plock** also allows these segments to be unlocked. The effective user id of the calling process must have the appropriate privilege to use this call.

plock performs the function specified by *op*:

PROCLOCK Lock text and data segments into memory (process lock).

TXTLOCK Lock text segment into memory (text lock).

DATLOCK Lock data segment into memory (data lock).

UNLOCK Remove locks.

Return Values

On success, **plock** returns 0. On failure, **plock** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **plock** fails and sets **errno** to:

EPERM The effective user id of the calling process does not have the appropriate privilege.

EINVAL *op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.

EINVAL *op* is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.

EINVAL *op* is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process.

EINVAL *op* is equal to **UNLOCK** and no lock exists on the calling process.

EAGAIN Not enough memory, or there is insufficient resources.

SEE ALSO

exec(BA_OS), **memcntl**(RT_OS)

FUTURE DIRECTIONS

plock is described in terms of text and data segments but a process address space is usually described as a collected of **mmap**ed objects.

LEVEL

Level 2.

NOTICES

memcntl is the preferred interface to memory locking.

Considerations for Threads Programming

Sibling threads share (by definition) the same address space; modifications to the address space by one can be perceived by the others.

NAME

priocntl – process scheduler control

SYNOPSIS

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/fppriocntl.h>
#include <sys/tspriocntl.h>

long priocntl(idtype_t idtype, id_t id, int cmd, void *arg);
```

DESCRIPTION

priocntl provides for control over the scheduling of active processes.

Processes fall into distinct classes with a separate scheduling policy applied to each class. The two classes currently supported are the fixed priority class and the time-sharing class. The characteristics of these classes are described under the corresponding headings below. The class attribute of a process is inherited across the **fork**(BA_OS) and **exec**(BA_OS) system calls. **priocntl** can be used to dynamically change the class and other scheduling parameters associated with a running process or set of processes given the appropriate permissions as explained below.

In the default configuration, the highest fixed priority process runs before any other process. Therefore, inappropriate use of fixed priority processes can have a dramatic negative impact on system performance.

For **priocntl**, the *idtype* and *id* arguments are used together to specify the set of processes. The interpretation of *id* depends on the value of *idtype*. The possible values for *idtype* and corresponding interpretations of *id* are as follows:

P_PID *id* is a process ID specifying a single process to which the **priocntl** system call is to apply.

P_PPID *id* is a parent process ID. The **priocntl** system call applies to all processes with the specified parent process ID.

P_PGID *id* is a process group ID. The **priocntl** system call applies to all processes in the specified process group.

P_SID *id* is a session ID. The **priocntl** system call applies to all processes in the specified session.

P_CID *id* is a class ID (returned by **priocntl PC_GETCID** as explained below). The **priocntl** system call applies to all processes in the specified class.

P_UID *id* is a user ID. The **priocntl** system call applies to all processes with this effective user ID.

P_GID *id* is a group ID. The **priocntl** system call applies to all processes with this effective group ID.

P_ALL The **priocntl** system call applies to all existing processes. The value of *id* is ignored. The permission restrictions described below still apply.

An *id* value of **P_MYID** can be used with the *idtype* value to specify the calling process's process ID, parent process ID, process group ID, session ID, class ID, user ID, or group ID.

To change the scheduling parameters of a process (using the **PC_SETPARMS** command as explained below) the real or effective user ID of the process calling **priocntl** must match the real or effective user ID of the receiving process or the calling process must have appropriate privilege. See the subsections below for details for each class. These are the minimum permission requirements enforced for all classes. An individual class may impose additional permissions requirements when setting processes to that class and/or when setting class-specific scheduling parameters.

A special **sys** scheduling class exists for scheduling the execution of certain special system processes (such as the swapper process). It is not possible to change the class of any process to **sys**. In addition, any processes in the **sys** class that are included in a specified set of processes are disregarded by **priocntl**. For example, an *idtype* of **P_UID** and an *id* value of zero would specify all processes with a user ID of zero except processes in the **sys** class and (if changing the parameters using **PC_SETPARMS**) the **init** process.

The **init** process is a special case. In order for a **priocntl** call to change the class or other scheduling parameters of the **init** process (process ID 1), it must be the only process specified by *idtype* and *id*. The **init** process may be assigned to any

The **pc_clparms** buffer holds class-specific scheduling parameters. The format of this parameter data for a particular class is described under the appropriate heading below. **PC_CLPARMSZ** is the length of the **pc_clparms** buffer and is defined in **sys/priocntl.h**.

Commands

Available **priocntl** commands are:

PC_GETCID

Get class ID and class attributes for a specific class given class name. The *idtype* and *id* arguments are ignored. If *arg* is non-null, it points to a structure of type **pcinfo_t**. The **pc_clname** buffer contains the name of the class whose attributes you are getting.

On success, the class ID is returned in **pc_cid**, the class attributes are returned in the **pc_clinfo** buffer, and the **priocntl** call returns the total number of classes configured in the system (including the **sys** class). If the class specified by **pc_clname** is invalid or is not currently configured the **priocntl** call returns –1 with **errno** set to **EINVAL**. The format of the attribute data returned for a given class is defined in the **sys/fppriocntl.h** or **sys/tspriocntl.h** header file and described under the appropriate heading below.

If *arg* is a **NULL** pointer, no attribute data is returned but the **priocntl** call still returns the number of configured classes.

PC_GETCLINFO

Get class name and class attributes for a specific class given class ID. The *idtype* and *id* arguments are ignored. If *arg* is non-null, it points to a structure of type **pcinfo_t**. **pc_cid** is the class ID of the class whose attributes you are getting.

On success, the class name is returned in the **pc_clname** buffer, the class attributes are returned in the **pc_clinfo** buffer, and the **priocntl** call returns the total number of classes configured in the system (including the **sys** class). The format of the attribute data returned for a given class is defined in the **sys/fppriocntl.h** or **sys/tspriocntl.h** header file and described under the appropriate heading below.

If *arg* is a **NULL** pointer, no attribute data is returned but the **priocntl** call still returns the number of configured classes.

PC_SETPARMS

Set the class and class-specific scheduling parameters of the specified process(es). *arg* points to a structure of type **pcparms_t**. **pc_cid** specifies the class you are setting and the **pc_clparms** buffer contains the class-specific parameters you are setting. The format of the class-specific parameter data is defined in the **sys/fppriocntl.h** or **sys/tspriocntl.h** header file and described under the appropriate class heading below.

When setting parameters for a set of processes, **priocntl** acts on the processes in the set in an implementation-specific order. If **priocntl** encounters an error for one or more of the target processes, it may or may not continue through the set of processes, depending on the error. If the error is related to permissions (**EPERM**), **priocntl** continues through the process set, resetting the parameters for all target processes for which the calling process has appropriate permissions. **priocntl** then returns –1 with **errno** set to **EPERM** to indicate that the

operation failed for one or more of the target processes. If **priocntl** encounters an error other than permissions, it does not continue through the set of target processes but returns the error immediately.

PC_GETPARMS
Get the class and/or class-specific scheduling parameters of a process. *arg* points the a structure of type **pcparms_t**.

If **pc_cid** specifies a configured class and a single process belonging to that class is specified by the *idtype* and *id* values or the **procset** structure, then the scheduling parameters of that process are returned in the **pc_clparms** buffer. If the process specified does not exist or does not belong to the specified class, the **priocntl** call returns –1 with **errno** set to **ESRCH**.

If **pc_cid** specifies a configured class and a set of processes is specified, the scheduling parameters of one of the specified processes belonging to the specified class are returned in the **pc_clparms** buffer and the **priocntl** call returns the process ID of the selected process. The criteria for selecting a process to return in this case is class dependent. If none of the specified processes exist or none of them belong to the specified class the **priocntl** call returns –1 with **errno** set to **ESRCH**.

If **pc_cid** is **PC_CLNULL** and a single process is specified the class of the specified process us returned in **pc_cid** and its scheduling parameters are returned in the **pc_clparms** buffer.

Fixed Priority Class

The fixed priority class provides a fixed priority preemptive scheduling policy for those processes requiring fast and deterministic response and absolute user/application control of scheduling priorities. If the fixed priority class is configured in the system it should have exclusive control of the highest range of scheduling priorities on the system. This ensures that a runnable fixed priority process is given CPU service before any process belonging to any other class.

The fixed priority class has a range of fixed priority (**fp_pri**) values that may be assigned to processes within the class. Fixed priorities range from 0 to *x*, where the value of *x* is configurable and can be determined for a specific installation by using the **priocntl PC_GETCID** or **PC_GETCLINFO** command.

The fixed priority scheduling policy is a fixed priority policy. The scheduling priority of a fixed priority process is never changed except as the result of an explicit request by the user/application to change the **fp_pri** value of the process.

For processes in the fixed priority class, the **fp_pri** value is, for all practical purposes, equivalent to the scheduling priority of the process. The **fp_pri** value completely determines the scheduling priority of a fixed priority process relative to other processes within its class. Numerically higher **fp_pri** values represent higher priorities. Since the fixed priority class controls the highest range of scheduling priorities in the system it is guaranteed that the runnable fixed priority process with the highest **fp_pri** value is always selected to run before any other process in the system.

In addition to providing control over priority, **priocntl** provides for control over the length of the time quantum allotted to processes in the fixed priority class. The time quantum value specifies the maximum amount of time a process may run assuming that it does not complete or enter a resource or event wait state (**sleep**). Note that if another process becomes runnable at a higher priority the currently running process may be preempted before receiving its full time quantum.

The system's process scheduler keeps the runnable fixed priority processes on a set of scheduling queues. There is a separate queue for each configured fixed priority and all fixed priority processes with a given **fp_pri** value are kept together on the appropriate queue. The processes on a given queue are ordered in FIFO order (that is, the process at the front of the queue has been waiting longest for service and receives the CPU first). Fixed priority processes that wake up after sleeping, processes that change to the fixed priority class from some other class, processes that have used their full time quantum, and runnable processes whose priority is reset by **priocntl** are all placed at the back of the appropriate queue for their priority. A process that is preempted by a higher priority process remains at the front of the queue (with whatever time is remaining in its time quantum) and runs before any other process at this priority. Following a **fork**(BA_OS) system call by a fixed priority process, the parent process continues to run while the child process (which inherits its parent's **fp_pri** value) is placed at the back of the queue.

Use the structure of **fpinfo_t**, defined in **sys/fppriocntl.h** which defines the format used for the attribute data for the fixed priority class.

```
short     fp_maxpri;     /* Maximum fixed priority */
```

The **priocntl PC_GETCID** and **PC_GETCLINFO** commands return fixed priority class attributes in the **pc_clinfo** buffer in this format.

fp_maxpri specifies the configured maximum **fp_pri** value for the fixed priority class (if **fp_maxpri** is *x*, the valid fixed priority priorities range from 0 to *x*).

The structure **fpparms_t** defined in **sys/fppriocntl.h** defines the format used to specify the fixed priority class-specific scheduling parameters of a process.

```
short     fp_pri;        /* Fixed priority */
ulong     fp_tqsecs;     /* Seconds in time quantum */
long      fp_tqnsecs;    /* Additional nanoseconds in quantum */
```

When using the **priocntl PC_SETPARMS** or **PC_GETPARMS** commands, if **pc_cid** specifies the fixed priority class, the data in the **pc_clparms** buffer is in this format.

The above commands can be used to set the fixed priority to the specified value or get the current **fp_pri** value. Setting the **fp_pri** value of a process that is currently running or runnable (not sleeping) causes the process to be placed at the back of the scheduling queue for the specified priority. The process is placed at the back of the appropriate queue regardless of whether the priority being set is different from the previous **fp_pri** value of the process. Note that a running process can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its **fp_pri** value to its current fixed priority value. To change the time quantum of a process without setting the priority or affecting the process's position on the queue, the **fp_pri** field should be set to the special value **FP_NOCHANGE** (defined in **sys/fppriocntl.h**). Specifying **FP_NOCHANGE** when changing the class of a process to fixed priority from some other class results in the

fixed priority being set to zero.

For the **priocntl PC_GETPARMS** command, if **pc_cid** specifies the fixed priority class and more than one fixed priority process is specified, the scheduling parameters of the fixed priority process with the highest **fp_pri** value among the specified processes are returned and the process ID of this process is returned by the **priocntl** call. If there is more than one process sharing the highest priority, the one returned is implementation-dependent.

The **fp_tqsecs** and **fp_tqnsecs** fields are used for getting or setting the time quantum associated with a process or group of processes. **fp_tqsecs** is the number of seconds in the time quantum and **fp_tqnsecs** is the number of additional nanoseconds in the quantum. For example setting **fp_tqsecs** to 2 and **fp_tqnsecs** to 500,000,000 (decimal) would result in a time quantum of two and one-half seconds. Specifying a value of 1,000,000,000 or greater in the **fp_tqnsecs** field results in an error return with **errno** set to **EINVAL**. Although the resolution of the **tq_nsecs** field is very fine, the specified time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. For example, the finest resolution currently available on a system is 10 milliseconds (1 ''tick''). Setting **fp_tqsecs** to 0 and **fp_tqnsecs** to 34,000,000 would specify a time quantum of 34 milliseconds, which would be rounded up to 4 ticks (40 milliseconds) on a machine with 10-millisecond resolution. The maximum time quantum that can be specified is implementation-specific and equal to **LONG_MAX** ticks (defined in **limits.h**). Requesting a quantum greater than this maximum results in an error return with **errno** set to **ERANGE** (although infinite quantums may be requested using a special value as explained below). Requesting a time quantum of zero (setting both **fp_tqsecs** and **fp_tqnsecs** to 0) results in an error return with **errno** set to **EINVAL**.

The **fp_tqnsecs** field can also be set to one of the following special values (defined in **sys/fppriocntl.h**), in which case the value of **fp_tqsecs** is ignored.

FP_TQINF	Set an infinite time quantum.
FP_TQDEF	Set the time quantum to the default for this priority
FP_NOCHANGE	Don't set the time quantum. This value is useful when you wish to change the fixed priority of a process without affecting the time quantum. Specifying this value when changing the class of a process to fixed priority from some other class is equivalent to specifying **FP_TQDEF**.

To change the class of a process to fixed priority (from any other class), or to change the priority or time quantum setting of a fixed priority process, the following conditions must be true:

The calling process must have the appropriate privilege.

The effective user ID of the calling process must match the effective user ID of the target process (or the calling process have the appropriate privilege).

The fixed priority and time quantum are inherited across the **fork**(BA_OS) and **exec**(BA_OS) system calls.

Time-Sharing Class

The time-sharing scheduling policy provides for a fair and effective allocation of the CPU resource among processes with varying CPU consumption characteristics. The objectives of the time-sharing policy are to provide good response time to interactive processes and good throughput to CPU-bound jobs while providing a degree of user/application control over scheduling.

The time-sharing class has a range of time-sharing user priority (see **ts_upri**) values that may be assigned to processes within the class. A **ts_upri** value of zero is defined as the default base priority for the time-sharing class. User priorities range from *–x* to *+x* where the value of *x* is configurable and can be determined for a specific installation by using the **priocntl PC_GETCID** or **PC_GETCLINFO** command.

The purpose of the user priority is to provide some degree of user/application control over the scheduling of processes in the time-sharing class. Raising or lowering the **ts_upri** value of a process in the time-sharing class raises or lowers the scheduling priority of the process. It is not guaranteed, however, that a process with a higher **ts_upri** value will run before one with a lower **ts_upri** value. This is because the **ts_upri** value is just one factor used to determine the scheduling priority of a time-sharing process. The system may dynamically adjust the internal scheduling priority of a time-sharing process based on other factors such as recent CPU usage.

In addition to the system-wide limits on user priority (returned by the **PC_GETCID** and **PC_GETCLINFO** commands) there is a per process user priority limit (see **ts_uprilim** below), which specifies the maximum **ts_upri** value that may be set for a given process; by default, **ts_uprilim** is zero.

The structure **tsinfo_t** (defined in **sys/tspriocntl.h**) defines the format used for the attribute data for the time-sharing class.

```
short     ts_maxupri;    /* Limits of user priority range */
```

The **priocntl PC_GETCID** and **PC_GETCLINFO** commands return time-sharing class attributes in the **pc_clinfo** buffer in this format.

ts_maxupri specifies the configured maximum user priority value for the time-sharing class. If **ts_maxupri** is *x*, the valid range for both user priorities and user priority limits is from *–x* to *+x*.

The structure **tsparms_t** defined in **sys/tspriocntl.h**, defines the format used to specify the time-sharing class-specific scheduling parameters of a process.

```
short    ts_uprilim;  /* Time-Sharing user priority limit */
short    ts_upri;     /* Time-Sharing user priority */
```

When using the **priocntl PC_SETPARMS** or **PC_GETPARMS** commands, if **pc_cid** specifies the time-sharing class, the data in the **pc_clparms** buffer is in this format.

For the **priocntl PC_GETPARMS** command, if **pc_cid** specifies the time-sharing class and more than one time-sharing process is specified, the scheduling parameters of the time-sharing process with the highest **ts_upri** value among the specified processes is returned and the process ID of this process is returned by the **priocntl** call. If there is more than one process sharing the highest user priority, the one returned is implementation-dependent.

Any time-sharing process may lower its own **ts_uprilim** (or that of another process with the same user ID).

If the priority of the target process is to be raised above its current value, or if the target process's **ts_uprilim** is to be raised above a value of 0, the following conditions must be true:

The calling process must have the appropriate privilege.

The effective user ID of the calling process must match the effective user ID of the target process (or the calling process have the appropriate privilege).

Attempts by an unprivileged user process to raise a **ts_uprilim** or set an initial **ts_uprilim** greater than zero fail with a return value of –1 and **errno** set to **EPERM**.

Any time-sharing process may set its own **ts_upri** (or that of another process with the same user ID) to any value less than or equal to the process's **ts_uprilim**. Attempts to set the **ts_upri** above the **ts_uprilim** (and/or set the **ts_uprilim** below the **ts_upri**) result in the **ts_upri** being set equal to the **ts_uprilim**.

Either of the **ts_uprilim** or **ts_upri** fields may be set to the special value **TS_NOCHANGE** (defined in **sys/tspriocntl.h**) to set one value without affecting the other. Specifying **TS_NOCHANGE** for the **ts_upri** when the **ts_uprilim** is being set to a value below the current **ts_upri** causes the **ts_upri** to be set equal to the **ts_uprilim** being set. Specifying **TS_NOCHANGE** for a parameter when changing the class of a process to time-sharing (from some other class) causes the parameter to be set to a default value. The default value for the **ts_uprilim** is **0** and the default for the **ts_upri** is to set it equal to the **ts_uprilim** which is being set.

The time-sharing user priority and user priority limit are inherited across the **fork** and **exec** system calls.

Return Values

Unless otherwise noted above, **priocntl** returns a value of 0 on success. On failure, **priocntl** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **priocntl** fails and sets **errno** to:

EPERM An attempt was made to change the system time-sharing or fixed priority defaults, and the calling process does not have appropriate privileges (respectively, for the two classes).

EPERM The effective user ID of the calling process does not match the effective user ID of the target process, and the calling process does not have the appropriate privilege.

EPERM An attempt was made to change the class of the target process to fixed priority (from any class) and the calling process does not have the appropriate privileges.

EPERM An attempt was made to change the priority of a fixed priority process and the calling process does not have the privileges.

EPERM An attempt was made to raise the priority of a time-sharing process, or raise the **ts_prilim** of the process above 0, and the calling process does not have the appropriate privilege.

EINVAL	The argument *cmd* was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.
ERANGE	The requested time quantum is out of range.
ESRCH	None of the specified processes exist.
EFAULT	All or part of the area pointed to by one of the data pointers is outside the process's address space.
ENOMEM	An attempt to change the class of a process failed because of insufficient memory.
EAGAIN	An attempt to change the class of a process failed because of insufficient resources other than memory (for example, class-specific kernel data structures).

FUTURE DIRECTIONS

Real Time Class is now uniformly called Fixed Priority Scheduling Class to better describe its characteristics.

SEE ALSO

exec(BA_OS), **fork**(BA_OS), **nice**(AS_CMD), **priocntl**(AU_CMD)

LEVEL

Level 1.

NAME

profil – execution time profile

SYNOPSIS

```
#include <unistd.h>

void profil(unsigned short *buff, unsigned int bufsiz,
     unsigned int offset, unsigned int scale);
```

DESCRIPTION

profil provides CPU-use statistics by profiling the amount of CPU time expended by a program. **profil** generates the statistics by creating an execution histogram for a current process. The histogram is defined for a specific region of program code to be profiled, and the identified region is logically broken up into a set of equal size subdivisions, each of which corresponds to a count in the histogram. With each clock tick, the current subdivision is identified and its corresponding histogram count is incremented. These counts establish a relative measure of how much time is being spent in each code subdivision. The resulting histogram counts for a profiled region can be used to identify those functions that consume a disproportionately high percentage of CPU time.

buff is a buffer of *bufsiz* bytes in which the histogram counts are stored in an array of **unsigned short int**.

offset, *scale*, and *bufsiz* specify the region to be profiled.

offset is effectively the start address of the region to be profiled.

scale, broadly speaking, is a contraction factor that indicates how much smaller the histogram buffer is than the region to be profiled. More precisely, *scale* is interpreted as an unsigned fixed-point fraction with the binary point implied on the left. Its value is the reciprocal of the number of bytes in a subdivision, per byte of histogram buffer. Since there are two bytes per histogram counter, the effective ratio of subdivision bytes per counter is one half the scale.

Profiling is turned off by giving a *scale* of **0** or **1**. It is rendered ineffective by giving a *bufsiz* of **0**. Profiling is turned off when an **exec** routine is executed, but remains on in both child and parent after a call to the **fork** routine. Profiling will be turned off if an update in *buff* would cause a memory fault.

scale can be computed as (*RATIO* * **0200000L**), where *RATIO* is the desired ratio of *bufsiz* to profiled region size, and has a value between 0 and 1. Qualitatively speaking, the closer *RATIO* is to 1, the higher the resolution of the profile information.

bufsiz can be computed as (*size_of_region_to_be_profiled* * *RATIO*).

SEE ALSO

monitor(SD_LIB), **prof**(SD_CMD)

LEVEL

Level 2: September 30, 1989

NOTICES

Profiling is turned off by giving a *scale* of 0 or 1, and is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an **exec**(BA_OS) is executed, but remains on in both child and parent processes after a **fork**(BA_OS). Profiling is turned off if a *buff* update would cause a memory fault.

Considerations for Threads Programming

Statistics are gathered at the process level and represent the combined usage of all contained threads.

NAME

ptrace – process trace

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

int ptrace(int request, pid_t pid, int addr, int data);
```

DESCRIPTION

ptrace allows a parent process to control the execution of a child process. Its primary use is for the implementation of breakpoint debugging [see **sdb**(SD_CMD)]. When **ptrace** is used, the child process behaves normally until it encounters a signal [see **signal**(BA_OS)], at which time it enters a stopped state and its parent is notified via the **wait**(BA_OS) system call. When the child is in the stopped state, its parent can examine and modify its ''core image'' using **ptrace**. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the action to be taken by **ptrace** and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state on receipt of a signal rather than the state specified by *func* [see **signal**(BA_OS)]. The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results ensue if the parent does not expect to trace the child.

the parent. On failure a value of –1 is returned to the parent process and the parent's **errno** is set to **EIO**.

6 With this request, a few entries in the child's user area can be written. *data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are implementation specific but might include the general registers and the condition codes of the Processor Status Word.

7 This request causes the child to resume execution. If the *data* argument is 0, the signal that caused the child to stop is canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. On success, the value of *data* is returned to the parent. This request fails if *data* is not 0 or a valid signal number, in which case a value of –1 is returned to the parent process and the parent's **errno** is set to **EIO**.

8 This request causes the child to terminate with the same consequences as **exit**(BA_OS).

9 This request is implementation dependent but if operative, it is used to request single stepping through the instructions of the child.

To forestall possible fraud, **ptrace** inhibits the set-user-ID facility on subsequent **exec**(BA_OS) calls. If a traced process calls **exec**(BA_OS), it stops before executing the first instruction of the new image showing signal **SIGTRAP**.

Return Values

Upon successful completion, return values are specific to the request type. Upon failure, the **ptrace** returns a value of **-1** and sets **errno** to indicate an error.

Errors

In the following conditions, **ptrace** fails and sets **errno** to:

EIO *request* is an illegal number.

ESRCH *pid* identifies a child that does not exist or has not executed a **ptrace** with request **0**.

SEE ALSO

signal(BA_OS), **wait**(BA_OS)

FUTURE DIRECTIONS

Replaced by **mmap**(). This will be removed in a future issue of the SVID.

LEVEL

Level 2, July 1992.

NAME

semctl – semaphore control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
     int val;
     struct semid_ds *buf;
     ushort *array;
};

int semctl(int semid, int semnum, int cmd, . . . /* union semun arg */);
```

DESCRIPTION

semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum*: The level of permission required for each operation is shown with each command. The symbolic names for the values of *cmd* are defined by the **<sys/sem.h>** header file.

GETVAL Return the value of **semval** Requires read permission.

SETVAL Set the value of **semval** to *arg***.val**. When this command is successfully executed, the **semadj** value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of **(int) sempid**. Requires read permission.

GETNCNT Return the value of **semncnt**. Requires read permission.

GETZCNT Return the value of **semzcnt**. Requires read permission.

The following *cmd*s return and set, respectively, every **semval** in the set of semaphores.

GETALL Place **semval**s into array pointed to by *arg***.array**. Requires read permission.

SETALL Set **semval**s according to the array pointed to by *arg***.array**. Requires alter permission. When this cmd is successfully executed, the **semadj** values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg***.buf**. Requires read permission.

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg***.buf**:

sem_perm.uid
sem_perm.gid

sem_perm.mode /* only access permission bits */

This command can be executed only by a process that has an effective user **ID** equal to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid* or to a process that has the appropriate privilege.

IPC_RMID Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This command can be executed only by a process that has an effective user **ID** equal to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid* or to a process that has the appropriate privilege.

Return Values

On success, **semctl** returns a value that depends on *cmd*:

GETVAL	the value of **semval**
GETPID	the value of **(int) sempid**
GETNCNT	the value of **semncnt**
GETZCNT	the value of **semzcnt**
all others	a value of 0

On failure, **semctl** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **semctl** fails and sets **errno** to:

EACCES Operation permission is denied to the calling process

EINVAL *semid* is not a valid semaphore identifier.

EINVAL *semnum* is less than 0 or greater than **sem_nsems**.

EINVAL *cmd* is not a valid command.

ENOSYS if the functionality is not supported by the implementation.

ERANGE *cmd* is **SETVAL** or **SETALL** and the value to which **semval** is to be set is greater than the system imposed maximum.

EPERM *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user **ID** of the calling process is not equal to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid* and the calling process does not have appropriate privilege.

EFAULT *arg*.**buf** points to an illegal address.

SEE ALSO

semget(KE_OS), **semop**(KE_OS)

FUTURE DIRECTIONS

This interface is designated Level 2 to encourage the use of the new functionality introduced in the SVID Fourth Edition. In the future this interface will be removed from the SVID. However the interface will continue to be part of the SVID while it is required by XPG4.

LEVEL

Level 2, July 1993.

NAME

semget – get set of semaphores

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

DESCRIPTION

semget returns the semaphore identifier associated with *key*. This identifier is accessible by any process in the system, subject to normal access restrictions and the permissions set with *semflg*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores are created for *key* if one of the following is true:

key is equal to **IPC_PRIVATE**.

key does not already have a semaphore identifier associated with it, and (*semflg*&**IPC_CREAT**) is true.

On creation, the data structure associated with the new semaphore identifier is initialized as follows:

sem_perm.cuid, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** are set equal to the effective user **ID** and effective group **ID**, respectively, of the calling process.

The access permission bits of **sem_perm.mode** are set equal to the access permission bits of *semflg*.

sem_nsems is set equal to the value of *nsems*.

sem_otime is set equal to 0 and **sem_ctime** is set equal to the current time.

Return Values

On success, **semget** returns a non-negative integer, namely a semaphore identifier. On failure, **semget** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **semget** fails and sets **errno** to:

EINVAL	*nsems* is either less than or equal to zero or greater than the system-imposed limit.
EACCES	A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted.
EINVAL	A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to zero.
ENOENT	A semaphore identifier does not exist for *key* and (*semflg*&**IPC_CREAT**) is false.

ENOSPC A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores or semaphore identifiers system wide would be exceeded.

EEXIST A semaphore identifier exists for *key* but both (*semflg***&IPC_CREAT**) and (*semflg***&IPC_EXCL**) are both true.

SEE ALSO

semctl(KE_OS), **semop**(KE_OS)

FUTURE DIRECTIONS

This interface is designated Level 2 to encourage the use of the new functionality introduced in the SVID Fourth Edition. In the future this interface will be removed from the SVID. However the interface will continue to be part of the SVID while it is required by XPG4.

LEVEL

Level 2, July 1993.

NAME

semop – semaphore operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

DESCRIPTION

semop is used to perform atomically an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *sops* is a pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
short   sem_num;   /* semaphore number */
short   sem_op;    /* semaphore operation */
short   sem_flg;   /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

sem_op specifies one of three semaphore operations as follows, depending on whether its value is negative, positive, or zero:

If *sem_op* is a negative integer, one of the following occurs: Requires alter permission.

If **semval** is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from **semval**. Also, if (*sem_flg***&SEM_UNDO**) is true, the absolute value of *sem_op* is added to the calling process's **semadj** value [see **exit**(BA_OS)] for the specified semaphore.

If **semval** is less than the absolute value of *sem_op* and (*sem_flg***&IPC_NOWAIT**) is true, **semop** returns immediately.

If **semval** is less than the absolute value of *sem_op* and (*sem_flg***&IPC_NOWAIT**) is false, **semop** increments the **semncnt** associated with the specified semaphore and suspends execution of the calling process until one of the following conditions occur.

semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from **semval** and, if (*sem_flg***&SEM_UNDO**) is true, the absolute value of *sem_op* is added to the calling process's **semadj** value for the specified semaphore.

The *semid* for which the calling process is awaiting action is removed from the system [see **semctl**(KE_OS)]. When this occurs, **errno** is set equal to **EIDRM**, and a value of –1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in **signal**(BA_OS).

If *sem_op* is a positive integer, the value of *sem_op* is added to **semval** and, if (*sem_flg***&SEM_UNDO**) is true, the value of *sem_op* is subtracted from the calling process's **semadj** value for the specified semaphore. Requires alter permission.

If *sem_op* is zero, one of the following occurs: Requires read permission.

If **semval** is zero, **semop** returns immediately.

If **semval** is not equal to zero and (*sem_flg***&IPC_NOWAIT**) is true, **semop** returns immediately.

If **semval** is not equal to zero and (*sem_flg***&IPC_NOWAIT**) is false, **semop** increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

semval becomes zero, at which time the value of **semzcnt** associated with the specified semaphore is decremented.

The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set equal to **EIDRM**, and a value of –1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of **semzcnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in **signal**(BA_OS).

Return Values

On success, **semop** returns 0, and the value of **sempid** for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

On failure, **semop** returns –1 and sets **errno** to identify the error.

Errors

In the following conditions, **semop** fails and sets **errno** to:

EINTR	**semop** returned due to receipt of a signal.
EIDRM	**semop** returned due to the removal of *semid* from the system.
EFBIG	*sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*. In this instance, the signal **SIGXFSZ** will not be generated. However, if file sizes are too big, the signal **SIGXFSZ** will be generated.
E2BIG	*nsops* is greater than the system-imposed maximum.
EACCES	Operation permission is denied to the calling process
EAGAIN	

ENOSPC	The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.
EINVAL	*semid* is not a valid semaphore identifier.
EINVAL	The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.
ERANGE	An operation would cause a **semval** to overflow the system-imposed limit.
ERANGE	An operation would cause a **semadj** value to overflow the system-imposed limit.
EFAULT	*sops* points to an illegal address.

SEE ALSO

exec(BA_OS), **exit**(BA_OS), **fork**(BA_OS), **semctl**(KE_OS), **semget**(KE_OS),

FUTURE DIRECTIONS

This interface is designated Level 2 to encourage the use of the new functionality introduced in the SVID Fourth Edition. In the future this interface will be removed from the SVID. However the interface will continue to be part of the SVID while it is required by XPG4.

LEVEL

Level 2, July 1993.

NOTICES

Considerations for Threads Programming

While one thread is blocked, siblings might still be executing.

The Threads Library provides another semaphore facility for the synchronization of multithreaded programs. See **semaphore**(3synch). That facility can also be used for synchronization between processes. See discussion of the **USYNC_PROCESS** flag.

NAME

shmctl — shared memory control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

DESCRIPTION

The function `shmctl()` provides a variety of shared memory control operations as specified by *cmd*. The following values for *cmd* are available:

`IPC_STAT` Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in the *Kernel Extension Definitions* chapter.

`IPC_SET` Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode
```

This *cmd* can only be executed by a process that has an effective user ID equal to either the value of `shm_perm.cuid` or `shm_perm.uid` (in the data structure associated with *shmid*) or by a process with appropriate privileges.

`IPC_RMID` Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either the value of `shm_perm.cuid` or `shm_perm.uid` (in the data structure associated with *shmid*) or by a process with appropriate privileges.

RETURN VALUE

Upon successful completion, the function `shmctl()` returns a value of `0`; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `shmctl()` fails and sets `errno` to:

`EINVAL` if the value of *shmid* is not a valid shared memory identifier; or the value of *cmd* is not a valid command.

`EACCES` if the argument *cmd* is equal to `IPC_STAT` and the calling process does not have read permission.

`EPERM` if the argument *cmd* is equal to `IPC_RMID` or `IPC_SET` and the process does not have appropriate privileges and is not equal to the value of `shm_perm.cuid` or `shm_perm.uid` (in the data structure associated with *shmid*).

ENOSYS if the functionality is not supported by the implementation.

SEE ALSO

shmget(KE_OS), shmop(KE_OS).

FUTURE DIRECTIONS

This interface is designated Level 2 to encourage the use of mmap which is the preferred interface for this functionality. In the future this interface will be removed from the SVID. However the interface will continue to be part of the SVID while it is required by XPG4.

LEVEL

Level 2, July 1992.

Optional: The function `shmctl()` may not be present in all implementations of the Kernel Extension.

NAME

shmget — get shared memory segment

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
```

DESCRIPTION

The function `shmget()` returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes are created for *key* if one of the following are true:

The argument *key* is equal to `IPC_PRIVATE`.

The argument *key* does not already have a shared memory identifier associated with it and (*shmflg* `&` `IPC_CREAT`) is true.

Upon creation, the data structure associated with the new shared memory-identifier is initialized as follows:

The value of `shm_perm.cuid` and `shm_perm.uid` are set equal to the effective user ID of the calling process.

The value of `shm_perm.cgid` and `shm_perm.gid` are set equal to the effective group ID of the calling process.

The access permission bits of `shm_perm.mode` are set equal to the access permission bits of *shmflg*.

The argument *shm_segsz* is set equal to the value of *size*.

The value of `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are set equal to `0`.

The value of `shm_ctime` is set equal to the current time.

RETURN VALUE

Upon successful completion, the function `shmget()` returns a non-negative integer, namely a shared memory identifier; otherwise, it returns a value of `-1` and sets `errno` to indicate an error.

ERRORS

Under the following conditions, the function `shmget()` fails and sets `errno` to:

`EINVAL` if the value of *size* is less than the system imposed minimum or greater than the system imposed maximum, or a shared memory identifier exists for the argument *key* but the size of the segment associated with it is less than *size* and *size* is not equal to `0`.

`EACCES` if a shared memory identifier exists for *key* but operation permission as specified by the access permission bits of *shmflg* would not be granted.

`ENOENT` if a shared memory identifier does not exist for the argument *key* and (*shmflg* `&` `IPC_CREAT`) is false.

`ENOSPC` if a shared memory identifier is to be created but the system imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.

`ENOSYS` if the functionality is not supported by the implementation.

`ENOMEM` if a shared memory identifier and associated shared memory segment are to be created, but the amount of available physical memory is not sufficient to fill the request.

`EEXIST` if a shared memory identifier exists for the argument *key* but `((`*shmflg* `& IPC_CREAT)&&(`*shmflg* `& IPC_EXCL))` is true.

SEE ALSO

shmctl(KE_OS), shmop(KE_OS).

FUTURE DIRECTIONS

This interface is designated Level 2 to encourage the use of mmap which is the preferred interface for this functionality. In the future this interface will be removed from the SVID. However the interface will continue to be part of the SVID while it is required by XPG4.

LEVEL

Level 2, July 1992.

Optional: The function `shmget()` may not be present in all implementations of the Kernel Extension.

NAME

shmop – shmat, shmdt — shared memory operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sysmacros.h>

void *shmat(int shmid, void *shmaddr, int shmflg);

int shmdt(void *shmaddr);
```

DESCRIPTION

The function `shmat()` attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to `(void *)0`, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to `(void *)0` and `(`*shmflg* `& SHM_RND)` is true, the segment is attached at the address given by `(`*shmaddr* `– (`*shmaddr* `% SHMLBA))`.

If *shmaddr* is not equal to `(void *)0` and `(`*shmflg* `& SHM_RND)` is false, the segment is attached at the address given by *shmaddr*.

The segment is attached for reading if `(`*shmflg* `& SHM_RDONLY)` is true and the calling process has read permission; otherwise, if it is not true and the calling process has read and write permission, the segment is attached for reading and writing.

The function `shmdt()` detaches from the calling process's data segments the shared memory segment located at the address specified by *shmaddr*.

The following symbolic names are defined by the `<sys/shm.h>` header file:

Name	*Description*
`SHMLBA`	segment low boundary address multiple
`SHM_RDONLY`	attach read-only (else read / write)
`SHM_RND`	round attach address to `SHMLBA`

RETURN VALUE

Upon successful completion, the function `shmat()` returns the data segment's start address of the attached shared memory segment. Upon successful completion, the function `shmdt()` returns a value of `0`. Otherwise, the functions `shmat()` and `shmdt()` return a value of `–1` and set `errno` to indicate an error.

ERRORS

Under the following conditions, the function `shmat()` fails and sets `errno` to:

`EACCES` if operation permission is denied to the calling process [see the *Kernel Extension Definitions* chapter].

`EMFILE` if the number of shared memory segments attached to the calling process would exceed the system impose limit.

`ENOMEM` if the available data space is not large enough to accommodate the shared memory segment.

`ENOSYS` if the functionality is not supported by the implementation.

`EINVAL` if the value of *shmid* is not a valid shared memory identifier; or the value of *shmaddr* is not equal to `0` and the value of (*shmaddr* – (*shmaddr* % `SHMLBA`)) is an illegal address; or the value of *shmaddr* is not equal to `0`, (*shmflg* & `SHM_RND`) is false and the value of *shmaddr* is an illegal address.

Under the following conditions, the function `shmdt()` fails (and does not detach the shared memory segment) and sets `errno` to:

`EINVAL` if *shmaddr* is not the start address of a shared memory segment.

SEE ALSO

exec(BA_OS), exit(BA_OS), fork(BA_OS), shmctl(KE_OS), shmget(KE_OS).

FUTURE DIRECTIONS

This interface is designated Level 2 to encourage the use of mmap which is the preferred interface for this functionality. In the future this interface will be removed from the SVID. However the interface will continue to be part of the SVID while it is present in by XPG4.

LEVEL

Level 2, July 1992.

Optional: the functions `shmat()` and `shmdt()` may not be present in all implementations of the Kernel Extension.