AT&T

System V Interface Definition

3

System V

Interface Definition

System V Interface Definition

System V
Interface Definition

Issue 2

Volume III

AT&T

ISBN 0-932764-10-X

Library of Congress Catalog Card No. 85-063224

Select Code No. 320-013

IMPORTANT NOTE TO USERS
While every effort has been made to ensure the accuracy of all information in this document, **AT&T** 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. **AT&T** 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. **AT&T** disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose.

AT&T 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.

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

This document was set on an **AUTOLOGIC, Inc. APS–5*** phototypesetter driven by the **troff** formatter operating on **UNIX**† System V on an **AT&T 3B20** computer.

----

* **UNIX** is a registered trademark of **AT&T**.
† **APS–5** is a trademark of AUTOLOGIC, Inc.

How to Order

To order copies of the *System V Interface Definition* by phone, you may call:

(800) 432-6600 (Inside U.S.A.)
(800) 255-1242 (Inside Canada)
(317) 352-8557 (Outside U.S.A. & Canada)

To order copies of the *System V interface Definition* by mail, write to:

AT&T Customer Information Center
Attn: Customer Service Representative
P.O. Box 19901
Indianapolis, IN 46219
U.S.A.

Be sure to include the address the books should be shipped to and a check or money order made payable to **AT&T**.

Please identify the books you want to order by Select Code. Select Codes for the *System V Interface Definition* are:

320-011	Volume I
320-012	Volume II
320-013	Volume III
307-131	Volumes I, II, and III

Table of Contents

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. The functionality of components is defined, but the implementation is not. 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* lets end-users and application-developers identify the features and functions available to them on any System V operating system.

Part I

A General Introduction to the System V Interface Definition

Chapter 1
General Introduction

1.1 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 building **C** language application-programs whose source-code must be portable from one System V environment to another. A system builder should also view these volumes as a necessary condition for supporting a System V environment that will host such applications.

This publication is intended to serve 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 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 is to facilitate the writing of application-program source-code that is directly portable across all System V implementations. Facilities outside of the Base System would require that the appropriate Extension be installed on the target environment.

1.2 STRUCTURE AND CONTENT

1.2.1 Partitioning into Base System and Extensions

The *System V Interface Definition* partitions System V components into a Base System and Extensions to that Base System. This does not change the definition of System V. It is instead a recognition 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, stand-alone 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, a system devoted to a single application, such as a vertically-integrated application package for managing a legal office, To execute, many applications programs will require only the components in the Base System. Other applications will need 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. The division between Base and Extensions will allow 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 to this the Advanced Utilities Extension. 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.

1.2.2 Conforming Systems

All conforming systems must support the source code interfaces and runtime behavior of all of 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.

This volume of the *System V Interface Definition* corresponds to functionality in **AT&T** System V Release 1.0, System V Release 2.0, and System V Release 3.0. An implementation of System V may conform to the System V Release 1.0 functionality, the System V Release 2.0 functionality, or the System V Release 3.0 functionality. All System V Release 2.0 enhancements to System V Release 1.0 are identified as such in the SVID; all System V Release 3.0 enhancements to System V Release 2.0 are identified as such in the SVID.

1.2.3 Organization of Technical Information

For ease of use, the SVID has been divided into several Volumes containing the following Extensions:

Volume 1. Base System

Kernel Extension

Volume 2. Basic Utilities Extension

Advanced Utilities Extension

Software Development Extension

Administered System Extension

Terminal Interface Extension

Volume 3. Base System Addendum

Terminal Interface Extension

Network Services Extension

Additional Volumes will define any further Extensions to System V.

The SVID defines the source-code interface and the run-time behavior of the components that make up 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 will be identified by a suffix of the form (XX_YYY) where XX identifies the Base System or the Extension that the component is in and YYY identifies the type of the component. For example, components defined in the Operating System Service Routines section of the Base System will be identified by (BA_OS), components defined in General Library Routines of the Base System will be identified by (BA_LIB), and components defined in the Operating System Service Routines section of the Kernel Extension will be identified by (KE_OS). Possible types are OS, LIB, CMD (commands or utilities) and ENV (environment).

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

An alphabetical index is provided in each Volume listing all components defined in that Volume. The index points to the detailed definition pages on which a component is to be found; the header for these pages may not contain the name of the component being sought. For example, in Volume I, the entry for the function **calloc** points to the MALLOC(BA_OS) pages, because the function **calloc** is defined with the function **malloc** on pages labeled MALLOC(BA_OS).

Each component definition follows the same structure. The sections are listed below; not all the following sections may be present in each description. If present, however, they will be in the given order. Sections entitled **EXAMPLE**, **APPLICATION USAGE**, 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 runtime behavior

- **RETURN VALUE** — value returned by the function
- **ERRORS** — possible error conditions
- **FILES** — names of files used
- **APPLICATION USAGE** or **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:

- All machine-specific or implementation-specific information has been removed. All implementation-specific constants have been replaced by symbolic names, which are defined in a separate section [see **implementation-specific constants** in **Volume I: Part II — Base System Definition: Chapter 4 — Definitions**]. When these symbolic names are used they always appear in curly brackets, e.g., {PROC_MAX}. The symbolic names correspond to those defined by the November 1985 draft of the **IEEE P1003** 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.
- A section entitled **FUTURE DIRECTIONS** has been added to selected component definitions. This section indicates how a component will evolve. The information ranges from specific changes in functionality to more general indications of proposed development.
- A section entitled **APPLICATION USAGE** or **USAGE** has been added to guide application developers on the expected or recommended usage of certain components. Detailed definitions of operating system services and library routines have an **APPLICATION USAGE** paragraph while utilities have a **USAGE** paragraph. While operating system services and library routines are only used by programs, utilities may be used by programs, by end-users or by system 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*, *system-administrator*, or *general*. The term *general* indicates that the utility might be used by all three: application-programs, end-users and system-administrators.

- A section entitled **LEVEL** defines each component's commitment level:

  Level-1 components will remain in the SVID and can be modified only in upwardly compatible ways. Any change in its definition will preserve the previous source-code interface and run-time behavior in order to ensure that the component remains upwardly-compatible.

  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. Level-2 components are labeled with the starting date of this three-year period.

1.3 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.
- Level-1 components will *not* move from one Extension into another Extension.
- 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.

1.4 C LANGUAGE DEFINITION

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

The following three references define the **C** language for System V Release 1.0, System V Release 2.0, and System V Release 3.0 respectively:

- *UNIX™ System V Programming Guide*, Issue 1, February 1982.
- *UNIX™ System V Programming Guide*, Issue 2, April 1984.
- *UNIX™ System V Programmers' Guide*, 1986.

Chapter 2
Future Directions

2.1 OPERATING SYSTEM STANDARDS

The **IEEE P1003** working group is currently pursuing a draft standard for a portable operating system interface. The *System V Interface Definition* is consistent with the trial-use standard (November 1985), with several minor exceptions. Full conformance to the **IEEE** standard will be strongly considered after its formal approval.

2.2 C LANGUAGE STANDARDIZATION

AT&T is committed to support the standardization of the **C** language being pursued by **ANSI X3J11**, in which its representatives take a leading role. Full conformance to the **ANSI** standard will be strongly considered after formal approval.

2.3 FLOATING POINT STANDARDS

The **IEEE P754** Standard for Binary Floating Point Arithmetic will be supported by System V. The existing library routines that deal with floating point numbers, and which are likely to change in order to support the **IEEE P754** Standard, belong to the following classes:

- routines that do arithmetic operations;
- routines that do input/output;
- routines that manipulate floating point numbers.

However, these changes are hardware dependent and will appear only on the machines whose underlying floating point data representation and exception handling mechanisms are those specified by the **IEEE P754** Standard.

2.4 GRAPHICS EXTENSION

This Extension will track current industry efforts to define standards for graphics functions. One area under active consideration is the Graphical Kernel Subsystem (GKS).

2.5 INTERNATIONALIZATION

Where necessary, modifications will be made, in an upwardly compatible way, to existing System V components to support internationalization. In addition, new components will be added to support features not currently available in System V. These will include tools that will allow *national supplements* to be added to an implementation of System V.

National supplements would be small packages that contained the necessary supplementary information, such as messages, databases, documentation, and device-drivers that, when installed, would allow an implementation of System V to process different national languages and support hardware (i.e., terminals, printers) and local conventions found in different countries. System builders would be able to create national supplements using the tools provided in System V.

More than one national supplement could be installed on a system at a time, resulting in a system with multiple language capabilities; however, national supplements are envisioned as self-contained, not requiring or depending on other installed national supplements.

Facilities that System V will provide to support internationalization and the development of national supplements are:

- Messages and text from the kernel, utilities, and application programs will be separated to enable support for national languages.

- Local conventions, or *environments*, will be supported transparently, depending on the language selected by the user. Among the conventions to be supported are date and time formats, collating sequences, and numeric representations.

- Supplementary code-sets will be supported to allow use of multiple code-sets, and consequently character symbols, in addition to the **ASCII** code-set.

- Sixteen-bit code-sets will be supported. This will allow languages of Far Eastern countries (i.e., Japan, Republic of China, Korea, the People's Republic of China, etc.) to be used.

- Language selection will be provided at the user-level to allow users of different languages to use the same system at the same time in their respective languages.

Message Handling.
In the future, System V will support a facility to produce messages and text in national languages. In conjunction with the *Error Handling Standards* defined in **Volume I: Part II — Base System Definition: Chapter 7 — General**

Library Routines, messages and text from the kernel, utilities, and applications would be stored separately. In addition, a set of administrative utilities would be provided to allow the creation of new messages and strings, as well as modification to existing ones.

Local Conventions.
Local conventions define the common forms and rules used to communicate information. The aim of internationalization is to provide System V applications and utilities with the capability to interact with the end-user according to these local-conventions. At the same time, applications and utilities must be portable and easily adapted to other conventions (i.e., they must be shielded from any particular set of conventions). Existing utilities and interfaces will be modified to support both implicit and explicit invocation of these conventions, with the following areas targeted for support:

Collating Sequence: The capability to define one or more *collating sequences* for a specific code-set will be provided. Utilities providing sorted output or requiring sorted input will be modified to allow invocation of different collating sequences. In addition, tools will be provided to support defining of specific collating sequences.

Character Classification: The capability to define, on a language-by-language basis, character classes will be provided. The CTYPE(BA_LIB) library will be enhanced to provide character classification in local languages. Where possible, this capability will be provided through the existing classification routines. In addition, new routines will be provided to support new capabilities (i.e., returning an indication of which code-set a particular character comes from).

Date and Time Format: The capability to enter and display date and time in the local language and according to local formats will be provided. This applies to all utilities or services that operate with date/time specifications.

Numeric Representation: The capability to define the rules for numeric editing (such as decimal delimiter) will be provided.

Currency Representation: The capability to specify rules and formats for editing local currency will be provided.

8th-bit Cleanup.
To support code-sets in addition to ASCII, all 8-bits of a byte will be used for character encoding. For example, some existing routines or utilities reject characters with octal values greater than **177**. Future releases will eliminate this and similar problems.

Code-Set and Character Support.
There are essentially two representations that make up the code-set:

the *external code-set* and the *internal code-set.*

The external code-set are those code-sets generated by input/output devices (i.e., terminals, printers, etc.). The most notable example is the seven-bit **ASCII**[1] code-set produced by most terminals and printers connected to System V today.

The internal code-set is a transformation of the external code-set according to the rules presented in this section, and is used to represent bytes throughout the rest of System V. Normally, no part of System V, except a device-driver, will see the external code-set; however, in many cases, the external and internal encodings will be the same with only minor exceptions.

The device-driver has the sole responsibility of mapping an external code-set to an internal code-set and vice-versa.

The following sections describe a template for transforming externally coded characters into internally coded characters, methods of designating a particular code-set to be used, and methods of designating a particular language to be used.

A *Code-Set Template* is a template for transforming externally coded characters into internally coded characters accessible by the System V operating system, utilities, and applications. The internal coding method discussed here is based on the **ISO 2022-1982** standard for code extension techniques, which suggests the following two techniques for shifting between code-sets:

- Single-shift
- Locking-shift

The single-shift is a single byte used to announce a temporary shift to another code-set. The byte, or bytes, immediately following the single-shift code are interpreted as part of a new code-set. Subsequent characters are interpreted as belonging to the primary code-set.

1. **ASCII**, as it is used here, is defined as the seven-bit code-set used for information interchange in the United States. It does *not* refer to the extended eight-bit **ASCII** code-set, sometimes known as **ASCII-8**, or local derivatives of the seven-bit **ASCII** code-set used in parts of Europe.

The **ISO** standard defines two single-shift characters:

1. **SS2**, or *single-shift two,* and
2. **SS3**, or *single-shift three.*

The **SS2** character is represented by hexadecimal **8e**, while the **SS3** character is represented by hexadecimal **8f**.

The locking-shift technique is used to temporarily *shift-in* and *shift-out* of code-sets. It consists of a pair of character sequences that allow a new code-set to be used for more than one character. While in the context of a locking-shift sequence, all characters, with the exception of single-shifted characters, are assumed to belong to the new code-set.

Because of the context sensitivity of the locking-shift sequence, this method is not recommended for use in System V. Therefore, the use of the single-shift sequence is recommended to reduce the context sensitivity to as little as possible.

In addition to using the single-shifts to distinguish characters, the eighth-bit will also be used to distinguish between the primary code-set and characters in one of the three supplementary code-sets. By using the combination of eighth-bit and single-shift characters, the internal coding method specifies a template for allowing four code-sets to coexist simultaneously: one primary code-set and three supplementary code-sets, with the two of the latter denoted by a single-shift character. The representations for these internal code-sets are shown below:

Code-Set	Internal Representation
Set 0 (Primary code-set)	0xxxxxxx
Set 1 (Supplementary code-set #1)	1xxxxxxx — or — 1xxxxxxx 1xxxxxxx
Set 2 (Supplementary code-set #2)	SS2 1xxxxxxx — or — SS2 1xxxxxxx 1xxxxxxx
Set 3 (Supplementary code-set #3)	SS3 1xxxxxxx — or — SS3 1xxxxxxx 1xxxxxxx

Designation of the exact value of the four code-sets is performed through a code-set designation and is discussed in the following section.

A *Code-Set Designation* will be dynamic and accessible/modifiable at the operating system, utility and application levels to satisfy the specific needs for supporting multiple code-sets. It will also reside at the file level, so files with different code-set designations can exist on the same machine. That is, one file may be encoded with one set of code-sets while another file is encoded with another set of code-sets.

Specifically, it is desirable for code-set designation to meet the following requirements:

1. Code-set designations should be supported at the file level. Each file would contain its own set of code-set designation values.
2. At file creation time, all files would be designated with a system-wide default value.
3. Code-set designations could be changed dynamically.
4. The code-set designation value should contain information about:
   - The width of a character in the code-set,
   - The specific code-set designated (e.g., **DIS 8859/1**[2], **JIS 6226**[3], etc.),
5. Code-set designation information should be transferrable with the file contents across networks.

In addition to the code-set designation, a *language-designation* would offer the ability to designate which of several languages should be used for producing systems messages and for establishing an overall profile of the user's environment. One method under consideration for this type of designation is to use one or more exported environment-variables. For example, a LANGUAGE variable would be used to denote the language (e.g., French, German, Italian, Japanese, English, etc.). This variable would also be used as an index to user profile information to determine which local conventions to use. The variable could be assigned at initiation of the login session and could also be changed at any time. In this way,

2. **DIS 8859/1** Latin Language no. 1 is the newly-adopted **ISO** standard code-set, supporting most of the Western European characters. It is an 8-bit code-set that contains **US ASCII** as a subset.

3. **JIS 6226** is a **ISO** standard code-set for supporting the Japanese language. It is a 16-bit code-set that contains both hiragana and katakana alphabets, as well as about 7000 of the kanji ideograms.

language-designation is performed at user-level and controls the language of all system messages and text coming out of the operating system, utilities and applications, as well as particular national conventions.

Handling Non-standard Code-Sets. There are several code-sets in the world that the code-set template described here cannot support. The problem centers around the use of the eighth-bit to distinguish between characters in different code-sets. Specifically, these code-sets are as follows:

- The *shifted*-**JIS** code-set used in Japan,
- The packed Hangul code-set used in Korea,
- The **Big 5** code-set used in the Republic of China (Taiwan),
- The Chinese Code for Data Communications also used in the Republic of China.

Present plans are to provide limited support for these code-sets. Limited support means that files containing these code-sets could be stored on System V machines. No other support is currently planned; this implies that the mechanism for processing these files would have to be built into applications.

Character Support. In some applications it will be necessary to manipulate the variable-width characters coming from the supplementary code-sets. Although some application developers may choose to develop their own facilities for supporting this, System V will provide a generic facility for manipulating internally coded eight-bit bytes to a data type that can represent characters in a consistent manner. Initially, a new data type will be defined in the **C** programming language to support up to 16-bits of information. In addition, routines that use this new data type will be provided to allow application-developers to perform operations on them.

Part II

Base System Definition Addendum

Chapter 3
Introduction

The Base System Addendum updates the Base System Definition of Issue 2, Volume 1 and serves to document new functionality introduced to System V in System V Release 3.0. New functions not available in earlier releases are identified by the symbols †† next to the function name at the top of the component page. Functions that existed previously but that have some additional functionality or other change as of Release 3.0 are documented with the changes marked by a vertical bar (¦) in the margin. These pages should be compared to their corresponding pages in the System V Interface Definition (SVID) Issue 2, Volume 1. Most of these changes are due to Future Directions now being included in the component definition. An appendix documents changes to Issue 2, Volume 1 that are due to error correction of the earlier SVID volume.

The Base System is intended to support 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 CHOWN(BA_OS) routine to change the owner of a file rather than using the CHOWN(AU_CMD) 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(BA_OS) 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. These 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) in **Volume I: Part III — Kernel Extension Definition: Chapter 10 — Environment.**

Definitions for the Base System are given in the next chapter, **Chapter 4 — Definitions** of SVID Issue 2, Volume 1. Because the Base System is a prerequisite for any Extension, these definitions also apply to the Extensions. **Chapter 5 — Environment** describes the Base System Environment, including error conditions, environmental variables, directory tree structure, data files and special device files that must be present on a Base System. **Chapter 6 — OS Service Routines** defines operating system service routines that provide applications access to basic system resources (e.g., allocating dynamic storage) and **Chapter 7 — General Library Routines** defines general purpose library routines (e.g., string handling routines). The remainder of this introduction gives an overview of the contents of **Chapter 6 — OS Service Routines** and **Chapter 7 — General Library Routines.**

3.1 OPERATING SYSTEM SERVICE ROUTINES

Table 3-1 lists the basic set of routines that provide operating system services, e.g., process control, to applications.

Table 3-1: Base System: OS Service Routines

abort	ABORT(BA_OS)	**opendir††**	DIRECTORY(BA_OS)
access	ACCESS(BA_OS)	**pause**	PAUSE(BA_OS)
alarm	ALARM(BA_OS)	**pclose**	POPEN(BA_OS)
calloc	MALLOC(BA_OS)	**pipe**	PIPE(BA_OS)
chdir	CHDIR(BA_OS)	**popen**	POPEN(BA_OS)
chmod	CHMOD(BA_OS)	**readdir††**	DIRECTORY(BA_OS)
chown	CHOWN(BA_OS)	**realloc**	MALLOC(BA_OS)
clearerr	FERROR(BA_OS)	**rewind**	FSEEK(BA_OS)
closedir††	DIRECTORY(BA_OS)	**rewinddir††**	DIRECTORY(BA_OS)
dup	DUP(BA_OS)	**rmdir††**	RMDIR(BA_OS)
dup2††	DUP2(BA_OS)	**setgid**	SETUID(BA_OS)
exit	EXIT(BA_OS)	**setpgrp**	SETPGRP(BA_OS)
fclose	FCLOSE(BA_OS)	**setuid**	SETUID(BA_OS)
fcntl	FCNTL(BA_OS)	**sighold††**	SIGSET(BA_OS)
fdopen	FOPEN(BA_OS)	**sigignore††**	SIGSET(BA_OS)
feof	FERROR(BA_OS)	**sigrelse††**	SIGSET(BA_OS)
ferror	FERROR(BA_OS)	**sigset††**	SIGSET(BA_OS)
fflush	FCLOSE(BA_OS)	**signal**	SIGNAL(BA_OS)
fileno	FERROR(BA_OS)	**sleep**	SLEEP(BA_OS)
fopen	FOPEN(BA_OS)	**stat**	STAT(BA_OS)
fread	FREAD(BA_OS)	**stime**	STIME(BA_OS)
free	MALLOC(BA_OS)	**system**	SYSTEM(BA_OS)
freopen	FOPEN(BA_OS)	**time**	TIME(BA_OS)
fseek	FSEEK(BA_OS)	**times**	TIMES(BA_OS)
fstat	STAT(BA_OS)	**ulimit**	ULIMIT(BA_OS)
ftell	FSEEK(BA_OS)	**umask**	UMASK(BA_OS)
fwrite	FREAD(BA_OS)	**uname**	UNAME(BA_OS)
getcwd	GETCWD(BA_OS)	**unlink**	UNLINK(BA_OS)
getegid	GETUID(BA_OS)	**ustat**	USTAT(BA_OS)
geteuid	GETUID(BA_OS)	**utime**	UTIME(BA_OS)
getgid	GETUID(BA_OS)	**wait**	WAIT(BA_OS)
getpgrp	GETPID(BA_OS)		
getpid	GETPID(BA_OS)		
getppid	GETPID(BA_OS)		
getuid	GETUID(BA_OS)		
ioctl	IOCTL(BA_OS)		
kill	KILL(BA_OS)		
link	LINK(BA_OS)		
lockf§	LOCKF(BA_OS)		
mallinfo†	MALLOC(BA_OS)		
malloc	MALLOC(BA_OS)		
mallopt†	MALLOC(BA_OS)		
mkdir††	MKDIR(BA_OS)		
mknod	MKNOD(BA_OS)		

close	CLOSE(BA_OS)	**fork**	FORK(BA_OS)
creat	CREAT(BA_OS)	**lseek**	LSEEK(BA_OS)
execl	EXEC(BA_OS)	**mount**	MOUNT(BA_OS)
execle	EXEC(BA_OS)	**open**	OPEN(BA_OS)
execlp	EXEC(BA_OS)	**read**	READ(BA_OS)
execv	EXEC(BA_OS)	**umount**	UMOUNT(BA_OS)
execve	EXEC(BA_OS)	**write**	WRITE(BA_OS)
execvp	EXEC(BA_OS)		

_exit	EXIT(BA_OS)	**sync**	SYNC(BA_OS)

The operating system service routines provide access to and control over system resources such as memory, files, process execution. Some System V routines that provide operating system services are not supported by the Base System. An application-program that used any of these would require an *extended* environment. See, for example, **Volume I: Part III — Kernel Extension Definition.**

All the routines in Table 3-1, except those marked with †, ††, or §, are common to System V Release 1.0, System V Release 2.0, and System V Release 3.0. Those marked with † first appeared in System V Release 2.0. The function **lockf**, marked with §, is a post System V Release 2.0 component. Those marked with †† first appeared in System V Release 3.0.

Table 3-1 is shown as three sets of routines, which reflect recommended usage by application-programs.

The first set of routines (from **abort** to **wait**) should fulfill the needs of most application-programs.

The second set of routines (from **close** to **write**) 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 **Chapter 4 — Definitions**] should be used instead of the routines **close**, **creat**, **lseek**, **open**, **read**, **write** (e.g., the *stdio* routine **fopen** should be used rather than the routine **open**).

The third set of routines (**_exit** and **sync**), although they are 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.

3.2 GENERAL LIBRARY ROUTINES

Table 3-2 lists the basic set of General Library Routines that are likely to be used by application-programs.

Table 3-2: Base System: General Library Routines

abs	ABS(BA_LIB)	**j0**	BESSEL(BA_LIB)
acos	TRIG(BA_LIB)	**j1**	BESSEL(BA_LIB)
asin	TRIG(BA_LIB)	**jn**	BESSEL(BA_LIB)
atan2	TRIG(BA_LIB)	**ldexp**	FREXP(BA_LIB)
atan	TRIG(BA_LIB)	**log10**	EXP(BA_LIB)
ceil	FLOOR(BA_LIB)	**log**	EXP(BA_LIB)
cos	TRIG(BA_LIB)	**matherr**	MATHERR(BA_LIB)
cosh	SINH(BA_LIB)	**modf**	FREXP(BA_LIB)
erf	ERF(BA_LIB)	**pow**	EXP(BA_LIB)
erfc	ERF(BA_LIB)	**sin**	TRIG(BA_LIB)
exp	EXP(BA_LIB)	**sinh**	SINH(BA_LIB)
fabs	FLOOR(BA_LIB)	**sqrt**	EXP(BA_LIB)
floor	FLOOR(BA_LIB)	**tan**	TRIG(BA_LIB)
fmod	FLOOR(BA_LIB)	**tanh**	SINH(BA_LIB)
frexp	FREXP(BA_LIB)	**y0**	BESSEL(BA_LIB)
gamma	GAMMA(BA_LIB)	**y1**	BESSEL(BA_LIB)
hypot	HYPOT(BA_LIB)	**yn**	BESSEL(BA_LIB)

_tolower	CONV(BA_LIB)	**memccpy**	MEMORY(BA_LIB)
_toupper	CONV(BA_LIB)	**memchr**	MEMORY(BA_LIB)
advance	REGEXP(BA_LIB)	**memcmp**	MEMORY(BA_LIB)
asctime	CTIME(BA_LIB)	**memcpy**	MEMORY(BA_LIB)
atof	STRTOD(BA_LIB)	**memset**	MEMORY(BA_LIB)
atoi	STRTOL(BA_LIB)	**setkey#**	CRYPT(BA_LIB)
atol	STRTOL(BA_LIB)	**step**	REGEXP(BA_LIB)
compile	REGEXP(BA_LIB)	**strcat**	STRING(BA_LIB)
crypt#	CRYPT(BA_LIB)	**strchr**	STRING(BA_LIB)
ctime	CTIME(BA_LIB)	**strcmp**	STRING(BA_LIB)
encrypt#	CRYPT(BA_LIB)	**strcpy**	STRING(BA_LIB)
gmtime	CTIME(BA_LIB)	**strcspn**	STRING(BA_LIB)
isalnum	CTYPE(BA_LIB)	**strdup††**	STRING(BA_LIB)
isalpha	CTYPE(BA_LIB)	**strlen**	STRING(BA_LIB)
isascii	CTYPE(BA_LIB)	**strncat**	STRING(BA_LIB)
iscntrl	CTYPE(BA_LIB)	**strncmp**	STRING(BA_LIB)
isdigit	CTYPE(BA_LIB)	**strncpy**	STRING(BA_LIB)
isgraph	CTYPE(BA_LIB)	**strpbrk**	STRING(BA_LIB)
islower	CTYPE(BA_LIB)	**strrchr**	STRING(BA_LIB)
isprint	CTYPE(BA_LIB)	**strspn**	STRING(BA_LIB)
ispunct	CTYPE(BA_LIB)	**strtod†**	STRTOD(BA_LIB)
isspace	CTYPE(BA_LIB)	**strtok**	STRING(BA_LIB)
isupper	CTYPE(BA_LIB)	**strtol**	STRTOL(BA_LIB)
isxdigit	CTYPE(BA_LIB)	**toascii**	CONV(BA_LIB)
localtime	CTIME(BA_LIB)	**tolower**	CONV(BA_LIB)
		toupper	CONV(BA_LIB)
		tzset	CTIME(BA_LIB)

bsearch	BSEARCH(BA_LIB)	**fgets**	GETS(BA_LIB)
clock	CLOCK(BA_LIB)	**fprintf**	PRINTF(BA_LIB)
ctermid	CTERMID(BA_LIB)	**fscanf**	SCANF(BA_LIB)
drand48	DRAND48(BA_LIB)	**fputc**	PUTC(BA_LIB)
erand48	DRAND48(BA_LIB)	**fputs**	PUTS(BA_LIB)
fgetc	GETC(BA_LIB)	**ftw**	FTW(BA_LIB)

getc	GETC(BA_LIB)	**perror***	PERROR(BA_LIB)
getchar	GETC(BA_LIB)	**printf**	PRINTF(BA_LIB)
getenv	GETENV(BA_LIB)	**putc**	PUTC(BA_LIB)
getopt	GETOPT(BA_LIB)	**putchar**	PUTC(BA_LIB)
gets	GETS(BA_LIB)	**putenv**†	PUTENV(BA_LIB)
getw	GETC(BA_LIB)	**puts**	PUTS(BA_LIB)
gsignal*	SSIGNAL(BA_LIB)	**putw**	PUTC(BA_LIB)
hcreate	HSEARCH(BA_LIB)	**qsort**	QSORT(BA_LIB)
hdestroy	HSEARCH(BA_LIB)	**rand**	RAND(BA_LIB)
hsearch	HSEARCH(BA_LIB)	**scanf**	SCANF(BA_LIB)
isatty	TTYNAME(BA_LIB)	**seed48**	DRAND48(BA_LIB)
jrand48	DRAND48(BA_LIB)	**setbuf**	SETBUF(BA_LIB)
lcong48	DRAND48(BA_LIB)	**setjmp**	SETJMP(BA_LIB)
lfind†	LSEARCH(BA_LIB)	**setvbuf**†	SETBUF(BA_LIB)
longjmp	SETJMP(BA_LIB)	**sprintf**	PRINTF(BA_LIB)
lrand48	DRAND48(BA_LIB)	**srand48**	DRAND48(BA_LIB)
lsearch	LSEARCH(BA_LIB)	**srand**	RAND(BA_LIB)
mktemp	MKTEMP(BA_LIB)	**sscanf**	SCANF(BA_LIB)
mrand48	DRAND48(BA_LIB)	**ssignal***	SSIGNAL(BA_LIB)
nrand48	DRAND48(BA_LIB)	**swab**	SWAB(BA_LIB)
		tdelete	TSEARCH(BA_LIB)
		tempnam	TMPNAM(BA_LIB)
		tfind†	TSEARCH(BA_LIB)
		tmpfile	TMPFILE(BA_LIB)
		tmpnam	TMPNAM(BA_LIB)
		tsearch	TSEARCH(BA_LIB)
		ttyname	TTYNAME(BA_LIB)
		twalk	TSEARCH(BA_LIB)
		ungetc	UNGETC(BA_LIB)
		vfprintf†	VPRINTF(BA_LIB)
		vprintf†	VPRINTF(BA_LIB)
		vsprintf†	VPRINTF(BA_LIB)

The general library routines perform a wide range of useful functions including: mathematical functions shown in the first part of Table 3-2; string and character handling routines shown in the second part of Table 3-2; I/O routines, search routines, sorting routines and others shown in the third part of Table 3-2.

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 [see **Volume II: Part V — Software Development Extension Definition**] is required to support the compilation of those application-programs.

Routines marked with † first appeared in System V Release 2.0. Routines marked with †† first appeared in System V Release 3.0. All others are in System V Release 1.0, System V Release 2.0, and System V Release 3.0. Routines marked with * are level-2, as defined in **Chapter 1 — General Introduction**. Routines marked with # are optional and may not be present on all conforming systems.

Chapter 4
Definitions

ASCII character set

Tables 4-1 and 4-2 are maps of the **ASCII** character set, giving octal and hexadecimal equivalents of each character. Although the **ASCII** code does not use the eighth-bit in an octet, this bit should not be used for other purposes because codes for other languages may need to use it (see section on **INTERNATIONALIZATION** in **Chapter 2 Future Directions**).

Table 4-1: 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

Table 4-2: 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

directory
Directories organize files into a hierarchical system of files 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 file name. 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 **path-name** of the root directory is / and the parent-directory of the root-directory is /.

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 (see below). 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(BA_OS) and SETGID(BA_OS) routines, respectively.

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

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 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 small integer used to identify a file for the purposes of doing I/O. The value of a file-descriptor is from **0** to {OPEN_MAX}**−1**. An open file-descriptor is obtained from a call to the CREAT(BA_OS), DUP(BA_OS), FCNTL(BA_OS), OPEN(BA_OS), or PIPE(BA_OS) routine. A process may have no more than {OPEN_MAX} file-descriptors open simultaneously.

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(BA_OS), WRITE(BA_OS), IOCTL(BA_OS), and CLOSE(BA_OS) routines.

file-name

Strings consisting of **1** to {NAME_MAX} characters may be used to name an ordinary 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 file-names because of the special meaning attached to these characters for file-name expansion by the command interpreter [see SYSTEM(BA_OS)]. Other characters to avoid are the hyphen, blank, tab, <, >, backslash, single and double quotes, accent grave, vertical bar, caret, curly braces, and parentheses. It is also advisable to avoid the use of non-printing characters in file names.

implementation-specific constants

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

In the SVID, for readability, these constants are replaced with 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 constants in its code. For example, an application-program would not be expected to test the length of an argument list given to an EXEC(BS_OS) routine to determine if it was greater than {ARG_MAX}. The following lists the implementation-specific constants that may be used in System V component definitions:

Name	*Description*
{ARG_MAX}	max. length of argument to **exec**
{CHAR_BIT}	number of bits in a **char**
{CHAR_MAX}	max. integer value of a **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**
{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**
{MAXDOUBLE}	max. decimal value of a **double**
{MAX_CHAR}	max. size of character input buffer
{NAME_MAX}	max. number of characters in a file-name
{OPEN_MAX}	max. number of files a process can have open
{PASS_MAX}	max. number of significant characters in a password
{PATH_MAX}	max. number of characters in a path-name
{PID_MAX}	max. value for a process-ID
{PIPE_BUF}	max. number bytes atomic in write to a pipe
{PIPE_MAX}	max. number of bytes written to a pipe in a write
{PROC_MAX}	max. number of simultaneous processes, system wide
{SHRT_MAX}	max. decimal value of a **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**
{UID_MAX}	max. value for a user-ID or group-ID
{USI_MAX}	max. decimal value of an **unsigned**
{WORD_BIT}	number of bits in a **word** or **int**
{CHAR_MIN}	min. integer value of a **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**

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)].

path-name and **path-prefix**
In a C program, a path-name 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 file-name. A null string is undefined and may be considered an error.

More precisely, a path-name is a null-terminated character-string as follows:

```
<path_name>::=<file_name>|<path_prefix><file_name>|/|.|..
<path_prefix>::=<rtprefix>|/<rtprefix>|empty
<rtprefix>::=<dirname>/|<rtprefix><dirname>/
```

where **<file_name>** is a string of **1** to {NAME_MAX} significant characters other than slash and null, and **<dirname>** is a string of **1** to {NAME_MAX} significant characters (other than slash and null) that names a directory. The result of names not produced by the grammar are undefined.

If a path-name begins with a slash, the path search begins at the root-directory. Otherwise, the search begins from the current-working-directory.

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

The meanings of **.** and **..** are defined under **directory**.

process-group-ID
Each active-process is a member of a process-group. The process-group is uniquely identified by a positive-integer, called the process-group-ID, which is the process-ID of the group-leader (see below). This grouping permits the signaling of related processes [see KILL(BA_OS)]. A process inherits the process-group-ID of the process that created it [see FORK(BA_OS) and EXEC(BA_OS)].

process-group-leader
A process-group-leader is any process whose process-group-ID is the same as its process-ID. Any process that is not a process-group-leader may detach itself from its current process-group and become a new process-group-leader by calling the SETPGRP(BA_OS) routine.

process-ID
Each active-process in the system is uniquely identified by a positive-integer called a process-ID. The range of this ID is from **0** to {PID_MAX}. By convention, process-ID **0** and **1** are reserved for special system-processes.

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(BA_OS) and SETGID(BA_OS) 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(BA_OS)].

special-processes
All special-processes are system-processes (e.g., a system's process-scheduler). At least process-IDs **0** and **1** 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 Standard I/O, *stdio* routines is shown below [see also the definition of **stdio-stream** below]. Detailed component definitions of each can be found in either **Chapter 6**, the system service (BA_OS) routines or **Chapter 7**, the general library (BA_LIB) routines.

(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 following *functions* are implemented as macros and must not be redeclared: **getc**, **getchar**, **putc**, **putchar**, **ferror**, **feof**, **clearerr**, and **fileno**. The macros **getc** and **putc** handle characters quickly. The macros **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.

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(BA_LIB) routine.

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 *stream.* A stream is a pointer to a type **FILE** defined by the <**stdio.h**> header file. The FOPEN(BA_OS) routine 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 file.
stdout	the standard output file.
stderr	the standard error file.

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 as soon as written; 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 new-line character is written or terminal input is requested). The SETBUF(BA_LIB) routines may be used to change the stream's buffering strategy.

super-user

A process is recognized as a super-user process and is granted special privileges if its effective-user-ID is **0**.

tty-group-ID

Each active-process can be a member of a terminal-group that shares a control terminal [see DEVTTY(BA_ENV)] and is identified by a positive-integer called the tty-group-ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group [see EXIT(BA_OS) and SIGNAL(BA_OS)].

Chapter 5
Environment

NAME

errors — error code and condition definitions

SYNOPSIS

#include <errno.h>

extern int errno;

DESCRIPTION

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

The component definitions given in **Chapter 6 — OS Service Routines** and **Chapter 7 — General Library Routines**, 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** will be set. The **errno** value **0** is reserved; no error condition will be 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 too long
An argument list longer than {ARG_MAX} bytes was presented to a member of the EXEC(BA_OS) family of routines.

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

EAGAIN Resource temporarily unavailable, try again later
For example, the FORK(BA_OS) routine failed because the system's process table is full.

EBADF Bad file number
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).

EBUSY Device or resource busy
An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable.

ECHILD No child processes
The WAIT(BA_OS) routine was executed by a process that had no existing or unwaited-for child processes.

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(BA_OS) routine).

EFAULT Bad address
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 a routine that takes a pointer argument is passed an invalid address, if the system can detect the condition. Because systems will 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 too large
The size of a file exceeded the maximum file size, {FCHR_MAX} [see ULIMIT(BA_OS)].

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 Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in a call to the SIGNAL(BA_OS) or KILL(BA_OS) routine). Also set by math routines.
EIO	I/O error Some physical I/O error has occurred. This error may, in some cases, occur on a call following the one to which it actually applies.
EISDIR	Is a directory An attempt was made to write on a directory.
ELIBACC	Reserved.
ELIBBAD	Reserved.
ELIBEXEC	Reserved.
ELIBMAX	Reserved.
ELIBSCN	Reserved.
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 to make more than the maximum number of links, {LINK_MAX}, to a file.
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 attempt was made to apply an inappropriate operation to a device (e.g., read a write-only device).
ENOENT	No such file or directory A file name is specified and the file should exist but doesn't, or one of the directories in a path-name does not exist, or a path-name is longer than {PATH_MAX} characters.
ENOEXEC	Exec format error A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid format.

ENOLCK No locks available
There are no more locks available. The system lock table is full.

ENOMEM Not enough space
During execution of an EXEC(BA_OS) routine, a program asks for more space than the system is able to supply. This is not a temporary condition; the maximum space size is a system parameter. 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(BA_OS) routine.

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

ENOTBLK Block device required
A non-block file was mentioned where a block device was required (e.g., in a call to the MOUNT(BA_OS) routine).

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

ENOTTY Not a character device
A call was made to the IOCTL(BA_OS) routine specifying a file that is not a special character 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.

EPERM No permission match
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

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

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

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

ESPIPE Illegal seek
A call to the LSEEK(BA_OS) routine was issued to a pipe.

ESRCH No such process
No process can be found corresponding to that specified by **pid** in the KILL(BA_OS) or PTRACE(KE_OS) routine.

ETXTBSY Text file busy
An attempt was 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.

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

APPLICATION USAGE

Because a few routines may not have an error return value, an application may set **errno** to zero, call the routine, and then check **errno** again to see if an error has occurred.

LEVEL

Level 1.

Chapter 6
OS Service Routines

NAME

abort — generate an abnormal process termination

SYNOPSIS

int abort()

DESCRIPTION

The function **abort** first closes all open files if possible, then causes the signal **SIGABRT** to be sent to the process. This invokes abnormal process termination routines, such as a core dump, which are implementation dependent.

APPLICATION USAGE

The signal sent by **abort**, **SIGABRT**, should not be caught or ignored by applications.

SEE ALSO

EXIT(BA_OS), SIGNAL(BA_OS), SIGSET(BA_OS).

LEVEL

Level 1.

NAME

access — determine accessibility of a file

SYNOPSIS

```
#include <unistd.h>

int access(path, amode)
char *path;
int amode;
```

DESCRIPTION

The function **access** checks the named file for either accessibility according to the bit-pattern contained in **amode**, or checks the named file for existence. In either case, the function **access** uses the real-user-ID in place of the effective-user-ID and the real-group-ID or equivalent in place of the effective-group-ID.

The argument **path** points to a path-name naming the file.

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 logical 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**.

When checking for accessibility, the owner of a file has permission checked with respect to the *owner* read, write, and execute mode bits. Members of the file's group other than the owner have permissions checked with respect to the *group* mode bits, and all others have permissions checked with respect to the *other* mode bits.

RETURN VALUE

If the requested access is permitted, the function **access** will return **0**; otherwise, it will return **−1** and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **access** will fail and will set **errno** to:

ENOTDIR	if a component of the path-prefix is not a directory.
ENOENT	if the named file does not exist.
ENOENT	if the path-name is longer than {PATH_MAX} characters.
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.
ETXTBSY	if write access is requested for a pure procedure (shared text) file that is being executed.

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

chmod — change mode of file

SYNOPSIS

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

int chmod(path, mode)
char *path;
int mode;
```

DESCRIPTION

The function **chmod** sets the access permission portion of the named file's mode according to the bit-pattern contained in the argument **mode**.

The argument **path** points to a path-name naming a file.

Symbolic constants defining the access permission bits are in the <**sys/stat.h**> header file and should be used to construct the argument **mode**. The value of the argument **mode** should be the logical OR of the values of the desired permissions:

Name	*Description*
S_ISUID	Set user-ID on execution.
S_ISGID	Set group-ID on execution.
S_ISVTX	Reserved.
S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IXUSR	Execute (search) by owner.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IXGRP	Execute (search) by group.
S_IROTH	Read by others (i.e., anyone else).
S_IWOTH	Write by others.
S_IXOTH	Execute (search) by others.

S_ENFMT Record locking enforced.

The effective-user-ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective-user-ID of the process is not super-user and the effective-group-ID of the process does not match the group-ID of the file, the access permission **S_ISGID** (set group-ID on execution) is cleared. This prevents an ordinary user from making itself an effective member of a group to which it does not belong. Similarly, the CHOWN(BA_OS) routine clears the set-user-ID and set-group-ID bits when invoked by other than the super-user.

For ordinary files, if the mode bit **S_ENFMT** (record locking enforced) is set and the mode bit **S_IXGRP** (execute or search by group) is not set, enforced record locking is enabled. This will affect future calls to OPEN(BA_OS), CREAT(BA_OS), READ(BA_OS) and WRITE(BA_OS) routines on this file.

RETURN VALUE

If successful, the function **chmod** will return **0**; otherwise, it will return **−1**, the file mode will be unchanged and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **chmod** will fail and will set **errno** to:

ENOTDIR if a component of the path-prefix is not a directory.

ENOENT if the named file does not exist.

ENOENT if the path-name is longer than {PATH_MAX} characters.

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 and the effective-user-ID is not super-user.

EROFS if the named file resides on a read-only file system.

SEE ALSO

CHOWN(BA_OS), MKNOD(BA_OS).

LEVEL

Level 1.

NAME

creat — create a new file or rewrite an existing one

SYNOPSIS

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

int creat(path, mode)
char *path;
int mode;

DESCRIPTION

The function **creat** creates a new ordinary file or prepares to rewrite an existing file named by the path-name pointed to by **path.**

If the file exists, the length is truncated to 0, the mode and owner are unchanged, and the file is open for writing [see **O_WRONLY** in OPEN(BA_OS)]. 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; and the access permission bits [see CHMOD(BA_OS)] of the file mode are set to the value of the argument **mode** modified as follows:

> The corresponding bits are ANDed with the complement of the process' file mode creation mask [see UMASK(BA_OS)]. Thus, the function **creat** clears each bit in the file mode whose corresponding bit in the file mode creation mask is set.

If successful, the function **creat** will return the file-descriptor and the file will be open for writing. A new file may be created with a **mode** that forbids writing. Even if the argument **mode** forbids writing, the function **creat** opens the file for writing.

Symbolic constants defining the access permission bits are specified in the **<sys/stat.h>** header file and should be used to construct **mode** [see CHMOD(BA_OS)].

The call **creat(path, mode)** is equivalent to the following [see OPEN(BA_OS)]:

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

The file-pointer is set to the beginning of the file. The file-descriptor is set to remain open across calls to the EXEC(BA_OS) routines [see FCNTL(BA_OS)]. No process may have more than {OPEN_MAX} files open simultaneously.

RETURN VALUE

If successful, the function **creat** will return a non-negative integer, namely the file-descriptor; otherwise, it will return **−1** and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **creat** will fail and will set **errno** to:

ENOTDIR	if a component of the path-prefix is not a directory.
ENOENT	if a component of the path-name should exist but does not.
ENOENT	if the path-name is longer than {PATH_MAX} characters.
EACCES	if a component of the path-prefix denies search permission.
EACCES	if the file does not exist and the directory in which the file is to be created does not permit writing.
EACCES	if the file exists and write permission is denied.
EROFS	if the named file resides or would reside on a read-only file system.
ETXTBSY	if the file is a pure procedure (shared text) file that is being executed.
EISDIR	if the named file is an existing directory.
EMFILE	if {OPEN_MAX} file-descriptors are currently open in the calling-process.
ENOSPC	if the directory to contain the file cannot be extended.
ENFILE	if the system file table is full.
EAGAIN	if the file exists with enforced record locking enabled and there are record-locks on the file [see CHMOD(BA_OS)].

APPLICATION USAGE

Normally, applications should use the *stdio* routines to open, close, read, and write files. In this case, the FOPEN(BA_OS) *stdio* routine should be used rather than the CREAT(BA_OS) routine.

SEE ALSO

CHMOD(BA_OS), CLOSE(BA_OS), DUP(BA_OS), FCNTL(BA_OS), LSEEK(BA_OS), OPEN(BA_OS), READ(BA_OS), UMASK(BA_OS), WRITE(BA_OS).

LEVEL

Level 1.

NAME

closedir, opendir, readdir, rewinddir — directory operations

SYNOPSIS

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

int closedir(dirp)
DIR *dirp;

DIR *opendir(filename)
char *filename;

struct dirent *readdir(dirp)
DIR *dirp;

void rewinddir(dirp)
DIR *dirp;
```

DESCRIPTION

The function **closedir** closes the directory-descriptor indicated by the argument **dirp** and frees the **DIR** structure associated with the directory-descriptor.

The function **opendir** opens the directory named by the argument **filename** and returns a pointer to the **DIR** structure associated with the directory.

The function **readdir** returns a pointer to a directory structure **dirent** that contains the next non-empty directory entry in the directory specified by the argument **dirp**. The structure **dirent** defined by the **<dirent.h>** header file describes a directory entry. It includes the inode number (**d_ino**) and the filename (**d_name**), which is a null-terminated string of at most {NAME_MAX} characters:

```
long   d_ino;        /* inode number of entry */
char   d_name[1];    /* name of file */
```

The function **rewinddir(dirp)** resets the position of the directory pointer specified by the argument **dirp** to the beginning of the directory.

RETURN VALUE

The function **opendir** returns a **NULL** pointer if **filename** cannot be accessed, or if **filename** is not a directory, or if enough memory to hold a **DIR** structure or a buffer for the directory entries cannot be allocated and **errno** indicates the error.

If successful, the function **readdir** returns a valid pointer. Upon reaching the end of the directory, the function **readdir** returns a **NULL** pointer. Otherwise, the function **readdir** returns a **NULL** pointer and **errno** indicates the error.

The function **closedir** returns **0** if successful; otherwise, it returns **−1** and **errno** indicates the error.

ERROR

Under the following conditions, the functions **closedir**, **opendir**, and **readdir** will fail and will set **errno** to:

opendir:

ENOTDIR	if a component of the path-prefix is not a directory.
EACCES	if a component of the path-prefix denies search permission.
EACCES	if read permission is denied for the specified directory.
EMFILE	if {OPEN_MAX} file- or directory-descriptors are currently open in this process.
ENOENT	if the path-name is longer than {PATH_MAX} characters.

readdir:

ENOENT	if the current directory-descriptor is not located at a valid entry.
EBADF	if **dirp** is not a valid open directory-descriptor.

closedir:

EBADF	if **dirp** is not a valid open directory-descriptor.

APPLICATION USAGE

The functions **closedir**, **opendir**, **readdir**, and **rewinddir** were added in System V Release 3.0.

EXAMPLE

The following sample code will search a directory for the entry *name*:

```
dirp = opendir(".");
while ((dp = readdir (dirp)) ! = NULL)
      if (strcmp(dp -> d_name, name) == 0) {
            closedir(dirp);
            return(FOUND);
      }
closedir(dirp);
return(NOT_FOUND);
```

LEVEL

Level 1.

NAME

dup2 — duplicate an open file-descriptor

SYNOPSIS

int dup2(fildes, fildes2)
int fildes, fildes2;

DESCRIPTION

The function **dup2** causes duplication of an open file-descriptor.

The argument **fildes** is an open file-descriptor [see **file-descriptor** in **Chapter 4 — Definitions**].

The argument **fildes2** is a non-negative integer less than {OPEN_MAX}.

The argument **fildes2** is set to refer to the same file as the argument **fildes**. If **fildes2** already refers to an open file, this file-descriptor is first closed.

RETURN VALUE

If successful, the function **dup2** will return a non-negative integer, namely the file-descriptor; otherwise, it will return **−1** and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **dup2** will fail and will set **errno** to:

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

EBADF if **fildes2** is negative or greater than or equal to {OPEN_MAX}.

APPLICATION USAGE

The function **dup2** was added in System V Release 3.0.

SEE ALSO

CREAT(BA_OS), CLOSE(BA_OS), DUP(BA_OS), EXEC(BA_OS), FCNTL(BA_OS), LOCKF(BA_OS), OPEN(BA_OS), PIPE(BA_OS).

LEVEL

Level 1.

NAME

execl, execv, execle, execve, execlp, execvp — execute a file

SYNOPSIS

int execl(path, arg0, arg1, ... argn, (char *)0)
char *path, *arg0, *arg1, ... *argn;

int execv(path, argv)
char *path, *argv[];

int execle(path, arg0, arg1, ... argn, (char *)0, envp)
char *path, *arg0, *arg1, ... *argn, *envp[];

int execve(path, argv, envp)
char *path, *argv[], *envp[];

int execlp(file, arg0, arg1, ... argn, (char *)0)
char *file, *arg0, *arg1, ... *argn;

int execvp(file, argv)
char *file, *argv[];

DESCRIPTION

All forms of the function **exec** transform the calling-process into a new process. The new process is constructed from an ordinary, executable file called the *new-process-file*. This file consists of a header, a text segment, and a data segment. There can be no return from a successful **exec** because the calling-process image is overlaid by the new process image.

When a **C** program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **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 argument **argc** is conventionally at least one and the initial member of the array **argv** points to a string containing the name of the file.

The argument **path** points to a path-name that identifies the new-process-file. For **execlp** and **execvp**, the argument **file** points to the new-process-file. The path-prefix for this file is obtained by a search of the directories passed as the *environment* line **PATH=** [see ENVVAR(BA_ENV) and SYSTEM(BA_OS)].

The arguments **arg0, arg1, ... argn** are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least **arg0** must be present and point to a string that is the same as **file** or **path** (or its last component).

The argument **argv** is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, **argv[0]** must point to a string that is the same as **file** or **path** (or its last component), and **argv** is terminated by a null pointer.

The argument **envp** is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process, and **envp** is terminated by a null-pointer. For **execl** and **execv**, a pointer to the environment of the calling-process is made available in the global cell:

```
extern char **environ;
```

and it is used to pass the environment of the calling-process to the new process.

The file-descriptors open in the calling-process remain open in the new process, except for those whose *close-on-exec* flag is set [see FCNTL(BA_OS)]. For those file-descriptors that remain open, the file-pointer is unchanged.

Signals set to the default action (**SIG_DFL**) in the calling-process will be set to the default action in the new process. Signals set to be ignored (**SIG_IGN**) by the calling-process will be ignored by the new process. Signals set to be held (**SIG_HOLD**) by the calling-process will be held by the new process. Signals set to be caught by the calling-process will be set to the default action in the new process [see SIGNAL(BA_OS) and SIGSET(BA_OS)].

If the set-user-ID-on-execution mode bit of the new-process-file is set, the **exec** sets the effective-user-ID of the new process to the owner-ID of the new-process-file [see CHMOD(BA_OS)]. Similarly, if the set-group-ID mode bit of the new-process-file is set, the effective-group-ID of the new process is set to the group-ID of the new-process-file. The real-user-ID and real-group-ID of the new process remain the same as those of the calling-process. The effective-user-ID and group-ID of the new process are saved for use by the SETUID(BA_OS) routine.

The new process also inherits at least the following attributes from the calling-process:

process-ID
parent-process-ID
process-group-ID
tty-group-ID [see EXIT(BA_OS), SIGNAL(BA_OS) and SIGSET(BA_OS)]
time left until an alarm clock signal [see ALARM(BA_OS)]
current-working-directory
root-directory
file mode creation mask [see UMASK(BA_OS)]
file size limit [see ULIMIT(BA_OS)]
utime, **stime**, **cutime**, and **cstime** [see TIMES(BA_OS)]
record-locks [see FCNTL(BA_OS) and LOCKF(BA_OS)]

RETURN VALUE

If the **exec** returns to the calling-process, an error has occurred; the **exec** will return **−1** and **errno** will indicate the error.

ERRORS

Under the following conditions, the **exec** will return to the calling-process and will set **errno** to:

ENOENT if one or more components of the path-name of the new-process-file do not exist.

ENOENT if the path-name is longer than {PATH_MAX} characters.

ENOTDIR if a component of the path-prefix of the new-process-file is not a directory.

EACCES if a directory in the new-process-file's path-prefix denies search permission, or if the new-process-file is not an ordinary file [see MKNOD(BA_OS)], or if the new-process-file's mode denies execution permission.

ENOEXEC if the **exec** is not an **execlp** or **execvp**, and the new-process-file has the appropriate access permission but is not a valid executable object.

ETXTBSY if the new-process-file is a pure procedure (shared text) file that is currently open for writing by some process.

ENOMEM if the new process image requires more memory than is allowed by the hardware or system-imposed maximum.

E2BIG if the number of bytes in the new process image's argument list exceeds the system-imposed limit of {ARG_MAX} bytes.

EFAULT if the new-process-file image is corrupted.

ELIBACC Reserved.

ELIBEXEC Reserved.

APPLICATION USAGE

Two interfaces for these functions are available. The list (**l**) versions: **execl**, **execle**, and **execlp** are useful when a known file with known arguments is being called. The arguments are the character-strings that are the file-name and the arguments. The variable (**v**) versions: **execv**, **execve**, and **execvp** are useful when the number of arguments is unknown in advance. The arguments are a file-name and a vector of strings containing the arguments.

If possible, applications should use the SYSTEM(BA_OS) routine, which is easier to use and supplies more functions, rather than the FORK(BA_OS) and EXEC(BA_OS) routines.

SEE ALSO

ALARM(BA_OS), EXIT(BA_OS), FORK(BA_OS), SIGNAL(BA_OS), SIGSET(BA_OS), TIMES(BA_OS), ULIMIT(BA_OS), UMASK(BA_OS).

LEVEL

Level 1.

NAME

fcntl — file control

SYNOPSIS

#include <fcntl.h>

int fcntl(fildes, cmd, arg)
int fildes, cmd;

DESCRIPTION

The function **fcntl** provides for control over open files.

The argument **fildes** is an open file-descriptor [see **file-descriptor** in **Chapter 4 — Definitions**].

The data type and value of **arg** are specific to the type of command specified by **cmd**. The symbolic names for commands and file status flags are defined by the **<fcntl.h>** header file.

The commands available are:

F_DUPFD Return a new file-descriptor as follows:

Lowest numbered available file-descriptor greater than or equal to the argument **arg**.

Same open file (or pipe) as the original file.

Same file-pointer as the original file (i.e., both file-descriptors share one file-pointer).

Same access-mode (*read, write,* or *read/write*) [see ACCESS(BA_OS)].

Same file status flags [see OPEN(BA_OS)].

The close-on-exec flag associated with the new file-descriptor is set to remain open across calls to the EXEC(BA_OS) routines.

F_GETFD Get the close-on-exec flag associated with the file-descriptor **fildes**. If the low-order bit is **0**, the file will remain open across calls to the EXEC(BA_OS) routines; otherwise, the file will be closed upon execution of any EXEC(BA_OS) routines.

F_SETFD Set the close-on-exec flag associated with **fildes** to the low-order bit of **arg** (**0** or **1** as above).

F_GETFL Get **file** status flags:
O_RDONLY, O_WRONLY, O_RDWR, O_NDELAY, O_APPEND, O_SYNC
[see OPEN(BA_OS)].

F_SETFL Set file status flags to **arg**. Only the flags **O_NDELAY, O_APPEND**, and **O_SYNC** may be set with **fcntl**.

The following commands are used for record-locking (see also **APPLICATION USAGE** below). Locks may be placed on an entire file or segments of a file.

F_GETLK Get the first lock which blocks the lock description given by the variable of type **struct flock** (see below) pointed to by **arg**. 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, then the structure is passed back unchanged except for the lock type which will be set to **F_UNLCK**.

NOTE: This command was added to **fcntl** following System V Release 1.0 and System V Release 2.0, and cannot be expected to be available in those releases.

F_SETLK Set or clear a file segment lock according to the variable of type **struct flock** (see below) pointed to by **arg**. **F_SETLK** is used to establish read (**F_RDLCK**) and write (**F_WRLCK**) locks, as well as remove either type of lock (**F_UNLCK**). **F_RDLCK, F_WRLCK**, and **F_UNLCK** are defined by the **<fcntl.h>** header file. If a read or write lock cannot be set, **fcntl** will return immediately with an error value of **−1**.

NOTE: This command was added to **fcntl** following System V Release 1.0 and System V Release 2.0, and cannot be expected to be available in those releases.

F_SETLKW This command is the same as **F_SETLK** except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

NOTE: This command was added to **fcntl** following System V Release 1.0 and System V Release 2.0, and cannot be expected to be available in those releases.

The structure **flock** defined by the **<fcntl.h>** header file describes a lock. It describes the type (**l_type**), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), and process-ID (**l_pid**):

```
short l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
short l_whence;  /* flag for starting offset */
long  l_start;   /* relative offset in bytes */
long  l_len;     /* if 0 then until EOF */
short l_pid;     /* returned with F_GETLK */
```

When a read-lock has been set on a segment of a file, other processes may also set read-locks on that segment or a portion of it. A read-lock prevents any other process from setting a write-lock on any portion of the protected area. 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 setting a read-lock or a write-lock on any portion of 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 value of **l_whence** is **0**, **1**, or **2** to indicate that the relative offset, **l_start** bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of **l_len** is the number of consecutive bytes to be locked. The process-ID **l_pid** field is only used with **F_GETLK** to return the value 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 zero (**0**). If such a lock also has **l_start** set to zero (**0**), the whole file will be locked.

Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments locked at each end of the originally locked segment. 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 after executing the FORK(BA_OS) routine.

If an ordinary file has enforced record locking enabled, then record-locks on the file will affect calls to CREAT(BA_OS), OPEN(BA_OS), READ(BA_OS), and WRITE(BA_OS).

RETURN VALUE

If successful, the function **fcntl** will return a value greater than or equal to zero that depends on **cmd** as follows:

F_DUPFD	a new file-descriptor.
F_GETFD	a value of flag (only the low-order bit is defined).
F_SETFD	a value other than **−1**.
F_GETFL	a value of file flags.
F_SETFL	a value other than **−1**.
F_GETLK	a value other than **−1**.
F_SETLK	a value other than **−1**.
F_SETLKW	a value other than **−1**.

If unsuccessful, the function **fcntl** will return **−1** and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **fcntl** will fail and will set **errno** to:

EBADF	if **fildes** is not a valid open file-descriptor.
EBADF	if **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	if **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.
EMFILE	if **cmd** is **F_DUPFD** and {OPEN_MAX} file-descriptors are currently open in the calling-process.
EINVAL	if **cmd** is **F_DUPFD** and **arg** is negative or greater than or equal to {OPEN_MAX}.
EINVAL	if **cmd** is **F_GETLK**, **F_SETLK**, or **F_SETLKW** and the data **arg** points to is not valid.

EACCES if **cmd** is **F_SETLK**, the type of lock (**l_type**) is a read-lock (**F_RDLCK**) or write-lock (**F_WRLCK**), and the segment of a file to be locked is already write-locked by another process, or the type is a write-lock and the segment of a file to be locked is already read-locked or write-locked by another process.

ENOLCK if **cmd** is **F_SETLK** or **F_SETLKW**, the type of lock is a read-lock or write-lock, and {LOCK_MAX} regions are already locked in the system.

EDEADLK if **cmd** is **F_SETLKW** and a deadlock condition was detected.

APPLICATION USAGE

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, for example:

```
...
flk->l_type = F_RDLCK;
if (fcntl(fd, F_SETLK, flk) == -1)
    if ((errno == EACCES) || (errno == EAGAIN))
        /*
         * section locked by another process,
         * check for either EAGAIN or EACCES
         * due to different implementations
         */
    else if ...
        /*
         * check for other errors
         */
```

The features of **fcntl** that deal with record locking are an update that followed System V Release 1.0 and System V Release 2.0.

SEE ALSO

CLOSE(BA_OS), EXEC(BA_OS), OPEN(BA_OS), LOCKF(BA_OS), READ(BA_OS), WRITE(BA_OS).

FUTURE DIRECTIONS

The error condition which currently sets **errno** to **EACCES** will instead set **errno** to **EAGAIN** [see also **APPLICATION USAGE** above].

LEVEL

Level 1.

NAME

fork — create a new process

SYNOPSIS

int fork()

DESCRIPTION

The function **fork** creates a new process. The new process (child-process) is a 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 (i.e., **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, *address*)
set-user-ID mode bit
set-group-ID mode bit
process-group-ID
tty-group-ID [see EXIT(BA_OS), SIGNAL(BA_OS) and SIGSET(BA_OS)]
current-working-directory
root-directory
file mode creation mask [see UMASK(BA_OS)]
file size limit [see ULIMIT(BA_OS)]

Additional attributes associated with an Extension to the Base System may be inherited from the parent-process [see, for example, **Part III — Kernel Extension Definition**].

The child-process differs from the parent-process as follows:

The child-process has a unique process-ID

The child-process has a different parent-process-ID (i.e., the process-ID of the parent-process).

The child-process has its own copy of the parent's file-descriptors. Each of the child-process' file-descriptors shares a common file-pointer with the corresponding file-descriptor of the parent-process.

The child-process' **utime**, **stime**, **cutime**, and **cstime** [see TIMES(BA_OS)] are set to **0**. The time left until an alarm clock signal is reset to **0**.

Record-locks set by the parent-process are not inherited by the child-process [see FCNTL(BA_OS) or LOCKF(BA_OS)].

RETURN VALUE

If successful, the function **fork** will return **0** to the child-process and will return the process-ID of the child-process to the parent-process; otherwise, it will return **−1** to the parent-process, no child-process will be created, and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **fork** will fail and will set **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.

ENOMEM if the process requires more space than the system is able to supply.

APPLICATION USAGE

The function **fork** creates a new process that is a copy of the calling-process and both processes will run as system resources become available. Because the goal is typically to create a new process that is *different* from the parent-process (i.e., the goal is to start a new program running), often the child-process immediately calls an EXEC(BA_OS) routine to transform itself and start the new program.

If possible, applications should use the SYSTEM(BA_OS) routine, which is easier to use and supplies more functions, rather than the FORK(BA_OS) and EXEC(BA_OS) routines.

SEE ALSO

ALARM(BA_OS), EXEC(BA_OS), FCNTL(BA_OS), LOCKF(BA_OS), SIGNAL(BA_OS), SIGSET(BA_OS), TIMES(BA_OS), ULIMIT(BA_OS), UMASK(BA_OS), WAIT(BA_OS).

LEVEL

Level 1.

NAME

fread, fwrite — buffered input/output

SYNOPSIS

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

int fread(ptr, size, nitems, stream)
char *ptr;
size_t size;
int nitems;
FILE *stream;

int fwrite(ptr, size, nitems, stream)
char *ptr;
size_t size;
int nitems;
FILE *stream;
```

DESCRIPTION

The function **fread** reads into an array pointed to by **ptr** up to **nitems** items of data from the named input **stream**, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length **size**. The function **fread** stops appending bytes if an end-of-file or error condition is encountered while reading **stream**, or if **nitems** items have been read. The function **fread** increments the data-pointer in **stream** to point to the byte following the last byte read if there is one [see FSEEK(BA_OS)]. The function **fread** does not change the contents of **stream**.

The function **fwrite** appends to the named output **stream** at most **nitems** items of data from the array pointed to by **ptr**. The function **fwrite** stops appending when it has appended **nitems** items of data or if an error condition is encountered on **stream**. The function **fwrite** does not change the contents of the array pointed to by **ptr**. The function **fwrite** increments the data-pointer in **stream** by the number of bytes written.

RETURN VALUE

If successful, both the function **fread** and the function **fwrite** will return the number of items read or written. If **size** or **nitems** is non-positive, no characters will be read or written, and both **fread** and **fwrite** will return **0**.

APPLICATION USAGE

The FERROR(BA_OS) or FEOF(BA_OS) routines must be used to distinguish between an error condition and an end-of-file condition.

SEE ALSO

FERROR(BA_OS), FOPEN(BA_OS), FSEEK(BA_OS), GETC(BA_LIB), GETS(BA_LIB), PRINTF(BA_LIB), PUTC(BA_LIB), PUTS(BA_LIB), READ(BA_OS), SCANF(BA_LIB). WRITE(BA_OS),

LEVEL

Level 1.

NAME

fseek, rewind, ftell — reposition a file-pointer in a stream

SYNOPSIS

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

int fseek(stream, offset, whence)
FILE *stream;
long offset;
int whence;

void rewind(stream)
FILE *stream;

long ftell(stream)
FILE *stream;
```

DESCRIPTION

The function **fseek** sets the position of the next input or output operation on the **stream**. 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 **<unistd.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 call **rewind(stream)** is equivalent to the following:

```
fseek(stream,0L,SEEK_SET)
```

except that the function **rewind** returns no value.

The functions **fseek** and **rewind** undo any effects of the UNGETC(BA_LIB) routine. 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 the named **stream**. The offset is always measured in bytes.

RETURN VALUE

The function **fseek** will return non-zero for improper seeks; otherwise, the function **fseek** will return zero. An improper seek is, for example, an **fseek** on a file that has not been opened via the FOPEN(BA_OS) routine; on a device incapable of seeking, such as a terminal; or on a stream opened via the POPEN(BA_OS) routine.

SEE ALSO

FOPEN(BA_OS), POPEN(BA_OS), UNGETC(BA_LIB).

LEVEL

Level 1.

NAME

lockf — record locking on files

SYNOPSIS

#include <unistd.h>

int lockf(fildes, function, size)
int fildes, function;
long size;

DESCRIPTION

NOTE: The function **lockf** first became available following System V Release 1.0 and System V Release 2.0.

The function **lockf** will allow sections of a file to be locked with advisory-mode or enforcement-mode locks depending on the mode of the file [see CHMOD(BA_OS)]. Calls to the function **lockf** from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates [see FCNTL(BA_OS) for more information about record-locking].

The argument **fildes** is an open file-descriptor. The file-descriptor must have been opened with write-only permission (**O_WRONLY**) or with read/write permission (**O_RDWR**) in order to establish a lock with this function call [see OPEN(BA_OS)].

The argument **function** is a control value which specifies the action to be taken. The permissible values for **function** are defined by the **<unistd.h>** header file as follows:

```
#define F_ULOCK 0 /* unlock locked sections */
#define F_LOCK  1 /* lock a section */
                  /* for exclusive use */
#define F_TLOCK 2 /* test and lock a section */
                  /* for exclusive use */
#define F_TEST  3 /* test section for locks */
                  /* by other processes */
```

F_TEST detects 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. All other values of **function** are reserved for future extensions and will result in an error return if they are not implemented.

The argument **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 or backward for a negative size (the preceding bytes up to but not including the current offset). If **size** is **0**, the section from the current offset through the largest file offset {FCHR_MAX} is locked (i.e., from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order 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. When this occurs, or if adjacent locked sections would occur, the sections are combined into a single locked 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 release (wholly or in part) one or more locked sections controlled by the process. Locked sections will be unlocked starting at the point of the file offset through **size** bytes or to the end of file if **size** is **0**. When all of a locked section is not released (i.e., the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section are still locked by the process. For example, releasing a center portion of a locked section will leave the portions of the section before and after it locked and requires an additional element in the table of active locks. If this table is full, an **EDEADLK** error is returned in **errno** and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process' locked resource. Thus calls to the function **lockf** or the FCNTL(BA_OS) routine scan for a deadlock prior to 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(BA_OS) routine may be used to provide a timeout facility in applications requiring it.

RETURN VALUE

If successful, the function **lockf** will return **0**; otherwise, it will return **−1** and **errno** will indicate the error.

ERRORS

The function **lockf** will fail and will set **errno** to:

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

EBADF if **function** is **F_LOCK** or **F_TLOCK** and **fildes** is not a valid file-descriptor open for writing.

EACCES if **function** is **F_TLOCK** or **F_TEST** and the section is already locked by another process.

EDEADLK if **function** is **F_LOCK** and a deadlock would occur; also if **function** is **F_LOCK**, **F_TLOCK**, or **F_ULOCK**, and {LOCK_MAX} regions are already locked in the system.

APPLICATION USAGE

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, for example:

```
...
if (lockf(fd, F_TLOCK, siz) == -1)
    if ((errno == EAGAIN) || (errno == EACCES))
        /*
         * section locked by another process
         * check for either EAGAIN or EACCES
         * due to different implementations
         */
    else if ...
        /*
         * check for other errors
         */
```

Record-locking should not be used in combination with the FOPEN(BA_OS), FREAD(BA_OS), FWRITE(BA_OS), etc., *stdio* routines. Instead, the more primitive, non-buffered routines (e.g., the OPEN(BA_OS) routine) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The *stdio* routines are the most common source of unexpected buffering.

SEE ALSO

CHMOD(BA_OS), CLOSE(BA_OS), CREAT(BA_OS), FCNTL(BA_OS), OPEN(BA_OS), READ(BA_OS), WRITE(BA_OS).

FUTURE DIRECTIONS

The error condition which currently sets **errno** to **EACCES** will instead set **errno** to **EAGAIN** [see also **APPLICATION USAGE** above].

LEVEL

Level 1.

NAME

lseek — move read/write file-pointer

SYNOPSIS

#include <unistd.h>

```
long lseek(fildes, offset, whence)
int fildes;
long offset;
int whence;
```

DESCRIPTION

The function **lseek** sets the file-pointer associated with **fildes** as specified by the value of the argument **whence**. Symbolic constants for **whence** are defined in the **<unistd.h>** header file:

Name	*Description*
SEEK_SET	set file-pointer equal to **offset** bytes.
SEEK_CUR	set file-pointer to current location plus **offset**.
SEEK_END	set file-pointer to **EOF** plus **offset**.

If successful, the function **lseek** returns the resulting pointer location, as measured in bytes from the beginning of the file. The function **lseek** modifies the file-pointer and does not affect the physical device.

The argument **fildes** is an open file-descriptor [see **file-descriptor** in **Chapter 4 — Definitions**].

RETURN VALUE

If successful, the function **lseek** will return a file-pointer value; otherwise, it will return **−1**, the file-pointer will remain unchanged and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **lseek** will fail and will set **errno** to:

EBADF	if **fildes** is not an open file-descriptor.
ESPIPE	if **fildes** is associated with a pipe or FIFO.
EINVAL	if **whence** is not **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**.

The significance of the file-pointer associated with a device incapable of seeking, such as a terminal, is undefined.

APPLICATION USAGE

Normally, applications should use the *stdio* routines to open, close, read, write, and manipulate files. Thus, an application that had used the FOPEN(BA_OS) *stdio* routine to open a file would use the FSEEK(BA_OS) *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), DUP(BA_OS), FCNTL(BA_OS), OPEN(BA_OS).

LEVEL

Level 1.

NAME

mkdir — make a directory

SYNOPSIS

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

int mkdir(path, mode)
char *path
int mode;
```

DESCRIPTION

The function **mkdir** creates a new directory.

The argument **path** specifies the name of the new directory.

The argument **mode** specifies the initial mode of the new directory. The protection bits of the argument **mode** are modified by the process' file mode creation mask [see UMASK(BA_OS)]. The value of the argument **mode** should be the logical OR of the values of the desired permissions:

Name	*Description*
S_IREAD	Read by owner.
S_IWRITE	Write by owner.
S_IEXEC	Execute (search) by owner.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IXGRP	Execute (search) by group.
S_IROTH	Read by others (i.e., anyone else).
S_IWOTH	Write by others.
S_IXOTH	Execute (search) by others.

The directory's owner ID is set to the process' effective-user-ID. The directory's group ID is set to the process' effective-group-ID. The newly created directory is empty, except for possible directory entries for "." (the directory itself) and ".." (the parent-directory) [see **directory** in **Chapter 4 — Definitions**].

RETURN VALUE

If successful, **mkdir** will return a value of **0**; otherwise, a value of **−1** is returned, no directory is created, and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **mkdir** will fail and will set **errno** to:

ENOTDIR if a component of the path-prefix is not a directory.

ENOENT if a component of the path-prefix does not exist.

ENOENT if the path-name is longer than {PATH_MAX} characters.

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 created.

EEXIST if the named path-name exists.

EROFS if the directory to be created is located on a read-only file system.

EMLINK if the maximum number of links to the parent directory, {LINK_MAX}, would be exceeded.

EIO if a physical I/O error has occurred.

ENOSPC if there is no free space available on the device containing the directory.

APPLICATION USAGE

The function **mkdir** was added in System V Release 3.0.

SEE ALSO

CHMOD(BA_OS), UMASK(BA_OS).

LEVEL

Level 1.

NAME

mknod — make a directory, a special or ordinary file, or a FIFO

SYNOPSIS

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

int mknod(path, mode, dev)
char *path;
int mode, dev;
```

DESCRIPTION

The function **mknod** creates a new file named by the path-name pointed to by the argument **path**.

The mode of the new file is initialized from the argument **mode**. Symbolic constants defining the value of the argument **mode** are in the **<sys/stat.h>** header file and should be used to construct **mode**. The value of the argument **mode** should be the logical OR of the values of the desired permissions:

Name	*Description*
S_IFMT	file type; one of the following:
S_IFIFO	FIFO-special
S_IFCHR	character-special
S_IFDIR	directory node
S_IFBLK	block-special
S_IFREG	ordinary-file
S_ISUID	set user-ID on execution
S_ISGID	set group-ID on execution
S_ISVTX	(reserved)
S_ENFMT	record locking enforced
S_IRUSR	read by owner
S_IWUSR	write by owner

S_IXUSR	execute (search) by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute (search) by group
S_IROTH	read by others (i.e., anyone else)
S_IWOTH	write by others
S_IXOTH	execute (search) 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.

Values of **mode** other than those above are undefined and should not be used. The owner, group, and other permission bits of **mode** are modified by the process' file mode creation mask: the function **mknod** clears each bit whose corresponding bit in the process' file mode creation mask is set [see UMASK(BA_OS)].

If the argument **mode** indicates a block-special or character-special file, the argument **dev** is a configuration-dependent specification of a character or block I/O device. The value of **dev** is obtained from the **st_rdev** field of the **stat** structure [see STAT(BA_OS)]. If **mode** does not indicate a block-special or character-special device, **dev** is ignored.

The function **mknod** may be invoked only by the super-user for file types other than FIFO-special.

RETURN VALUE

If successful, the function **mknod** will return **0**; otherwise, it will return **−1**, the new file will not be created, and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **mknod** will fail and will set **errno** to:

EPERM	if the effective-user-ID of the process is not super-user and the file type is not FIFO-special.
ENOTDIR	if a component of the path-prefix is not a directory.
ENOENT	if a component of the path-prefix does not exist.
ENOENT	if the path-name is longer than {PATH_MAX} characters.

EACCES if a component of the path-prefix denies search permission and the effective-user-ID of the process is not super-user.

EROFS if the directory in which the file is to be created is located on a read-only file system.

EEXIST if the named file exists.

ENOSPC if the directory to contain the new file cannot be extended.

APPLICATION USAGE

Normally, applications should use the MKDIR(BA_OS) routine to make a directory, since the function **mknod** may not establish directory entries for "." (the directory itself) and ".." (the parent-directory) [see **directory** in **Chapter 4 — Definitions**] and super-user privilege is not required.

SEE ALSO

CHMOD(BA_OS), EXEC(BA_OS), STAT(BA_OS), UMASK(BA_OS).

LEVEL

Level 1.

NAME

open — open file for reading or writing

SYNOPSIS

#include <fcntl.h>

int open(path, oflag [, mode])
char *path;
int oflag, mode;

DESCRIPTION

The function **open** opens a file-descriptor for the named file.

The argument **path** points to a path-name naming a file.

The function **open** sets the file status flags according to the value of the argument **oflag**. Symbolic names of flags are defined by the **<fcntl.h>** header file. The values of **oflag** are constructed by ORing 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_NDELAY This flag will 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_NDELAY** 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_NDELAY** 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 communication line:

If **O_NDELAY** is set:

The **open** will return without waiting for carrier.

If **O_NDELAY** is clear:

The **open** will block until carrier is present.

O_APPEND If set, the file-pointer will be set to the end of the file prior to each write.

O_SYNC If opening an ordinary file, this flag will affect subsequent writes. Each write [see WRITE(BA_OS)] should wait for both the file data and file status to be physically updated.

O_CREAT If the file does not exist, it is created, 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, and the access permission bits [see CHMOD(BA_OS)] of the file mode are set to the value of **mode** modified as follows [see CREAT(BA_OS)]:

The corresponding bits are ANDed with the complement of the process' file mode creation mask [see UMASK(BA_OS)]. Thus, the function **open** clears each bit in the file mode whose corresponding bit in the file mode creation mask is set.

Otherwise, if the file exists and **O_EXCL** is not set, this flag has no effect.

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

O_EXCL If **O_CREAT** is set and the file exists, the function **open** will fail.

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 calls to the EXEC(BA_OS) routines [see FCNTL(BA_OS)].

RETURN VALUE

If successful, the function **open** will return an open file-descriptor; otherwise, it will return **−1** and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **open** will fail and will set **errno** to:

ENOTDIR	if a component of the path-prefix is not a directory.
ENOENT	if **O_CREAT** is not set and the named file does not exist.
ENOENT	if a component of the path-name should exist but does not.
ENOENT	if the path-name is longer than {PATH_MAX} characters.
EACCES	if a component of the path-prefix denies search permission.
EACCES	if **O_CREAT** is set, the file does not exist, and the directory that would contain the file does not permit writing.
EACCES	if the **oflag** permission is denied for the named file.
EISDIR	if the named file is a directory and the **oflag** permission is write or read/write.
EROFS	if the named file resides on a read-only file system and the **oflag** permission is write or read/write.
EMFILE	if {OPEN_MAX} file-descriptors are currently open in this process.
ENXIO	if the named file is a character-special or block-special file and the device associated with this special file does not exist; or if **O_NDELAY** is set, the named file is a FIFO, **O_WRONLY** is set and no process has the file open for reading.
ETXTBSY	if the file is a pure procedure (shared text) file that is being executed and **oflag** specifies write or read/write permission.
EEXIST	if **O_CREAT** and **O_EXCL** are set, and the named file exists.
EINTR	if a signal was caught during the **open** operation.
ENFILE	if the system file table is full, {SYS_OPEN} files are open in the system.

ENOSPC if the directory to contain the file cannot be extended, the file does not exist, and **O_CREAT** is specified.

EAGAIN if the file exists with enforced record locking enabled, there are record-locks on the file [see CHMOD(BA_OS)], and **O_TRUNC** is specified.

APPLICATION USAGE

Normally, applications should use the *stdio* routines to open, close, read and write files. Thus, applications should use the FOPEN(BA_OS) *stdio* routine rather than using the OPEN(BA_OS) routine.

SEE ALSO

CLOSE(BA_OS), CREAT(BA_OS), DUP(BA_OS), FCNTL(BA_OS), LSEEK(BA_OS), READ(BA_OS), WRITE(BA_OS).

LEVEL

Level 1.

NAME

read — read from file

SYNOPSIS

```
int read(fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

The function **read** attempts to read **nbyte** bytes from the file associated with **fildes** into the buffer pointed to by **buf**.

The argument **fildes** is an open file-descriptor [see **file-descriptor** in **Chapter 4 — Definitions**].

On devices capable of seeking, the **read** starts at a position in the file given by the file-pointer associated with **fildes**. Upon return from the function **read**, the file-pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking, such as terminals, always read from the current position. The value of a file-pointer associated with such a file is undefined.

If successful, the function **read** will return the number of bytes read and placed in the buffer; this number may be less than **nbyte** if the file is associated with a communication line [see IOCTL(BA_OS) and TERMIO(BA_ENV)], or if the number of bytes left in the file is less than **nbyte** bytes, or if the file is a pipe or a special file. When an end-of-file has been reached, the function **read** will return **0**.

When attempting to read from an ordinary file with enforced record locking enabled [see CHMOD(BA_OS)], and all or part of the file to be read has a write-lock owned by another process (i.e., a blocking write-lock):

> If **O_NDELAY** is set, the function **read** will return **−1** and **errno** will be set to **EAGAIN**.
>
> If **O_NDELAY** is clear, the function **read** will sleep until all blocking write-locks are removed, or the function **read** is terminated by a signal.

When attempting to read from an empty pipe (or FIFO):

> If the pipe is no longer open for writing, **0** will be returned indicating end-of-file. Otherwise,
>
> > if **O_NDELAY** is clear, the read will block until data is written to the pipe or the pipe is no longer open for writing.
> >
> > if **O_NDELAY** is set, **0** will be returned.

When attempting to read a file associated with a character-special file that has no data currently available:

> If **O_NDELAY** is clear, the read will block until data becomes available.
>
> If **O_NDELAY** is set, **0** will be returned.

The function **read** reads data previously written to a file. If any portion of an ordinary file prior to the end-of-file has not been written, the function **read** returns bytes with value **0**. For example, the LSEEK(BA_OS) routine allows the file-pointer to be set beyond the end of 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 newly written data will return bytes with value **0** until data are written into the gap.

RETURN VALUE

If successful, the function **read** will return a non-negative integer indicating the number of bytes actually read; otherwise, it will return **−1** and **errno** will indicate the error.

ERRORS

The function **read** will fail and will set **errno** to:

EBADF if **fildes** is not a valid file-descriptor open for reading.

EINTR if a signal was caught during the **read** operation.

EIO if a physical I/O error has occurred.

ENXIO if the device associated with the file-descriptor is a block-special or character-special file and the value of the file-pointer is out of range.

EAGAIN if enforced record locking was enabled, **O_NDELAY** was set, and there was a write-lock owned by another process.

ENOLCK if {LOCK_MAX} regions are already locked in the system.

EDEADLK if **O_NDELAY** is clear and a deadlock condition was detected.

APPLICATION USAGE

Normally, applications should use the *stdio* routines to open, close, read and write files. Thus, an application that used the FOPEN(BA_OS) *stdio* routine to open a file should use the FREAD(BA_OS) *stdio* routine rather than the READ(BA_OS) routine to read it.

When **O_NDELAY** is set, portable application-programs should test for two conditions to determine that no data is currently available, for example:

```
fildes = open(path, O_RDONLY | O_NDELAY);
ret = read(fildes, buf, nbyte);
if (ret == 0 || (ret == -1 && errno == EAGAIN)) {
        .
        .   /* Data not available now. */
        .
}
```

SEE ALSO

CREAT(BA_OS), DUP(BA_OS), FCNTL(BA_OS), IOCTL(BA_OS), OPEN(BA_OS), POPEN(BA_OS).

FUTURE DIRECTIONS

When no data are present at the time of the read, the function **read** on a pipe, FIFO, or *tty-line* with the **O_NDELAY** flag set will return **−1**, rather than **0**, and **errno** will be set to **EAGAIN**.

LEVEL

Level 1.

NAME

rmdir — remove a directory

SYNOPSIS

```
int rmdir(path)
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, possibly, "." (the directory itself) and ".." (the parent-directory) [see **directory** in **Chapter 4 — Definitions**].

RETURN VALUE

If successful, **rmdir** will return a value of **0**; otherwise, a value of **−1** is returned, and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **rmdir** will fail and will set **errno** to:

EEXIST	if the directory to be removed contains directory entries other than "." and "..".
ENOTDIR	if a component of the path-prefix is not a directory.
ENOENT	if the named directory does not exist.
ENOENT	if the path-name is longer than {PATH_MAX} characters.
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.

APPLICATION USAGE

The function **rmdir** was added in System V Release 3.0.

SEE ALSO

MKDIR(BA_OS).

LEVEL

Level 1.

NAME

setuid, setgid — set user-ID and group-IDs

SYNOPSIS

int setuid(uid)
int uid;

int setgid(gid)
int gid;

DESCRIPTION

The function **setuid** is used to set the real-user-ID and effective-user-ID of the calling-process.

If the effective-user-ID of the calling-process is super-user, the real-user-ID, effective-user-ID, and the saved set-user-ID are set to **uid**.

If the effective-user-ID of the calling-process is not super-user, but its real-user-ID is equal to **uid**, the effective-user-ID is set to **uid**.

If the effective-user-ID of the calling-process is not super-user, but the saved set-user-ID from an EXEC(BA_OS) routine is equal to **uid**, the effective-user-ID is set to **uid**.

The function **setgid** is used to set the real-group-ID and effective-group-ID of the calling-process.

If the effective-user-ID of the calling-process is super-user, the real-group-ID and effective-group-ID are set to **gid**.

If the effective-user-ID of the calling-process is not super-user, but its real-group-ID is equal to **gid**, the effective-group-ID is set to **gid**.

If the effective-user-ID of the calling-process is not super-user, but the saved set-group-ID from an EXEC(BA_OS) routine is equal to **gid**, the effective-group-ID is set to **gid**.

RETURN VALUE

If successful, the function **setuid** will return **0**; otherwise, it will return **−1** and **errno** will indicate the error.

If successful, the function **setgid** will return **0**; otherwise, it will return **−1** and **errno** will indicate the error.

ERRORS

The function **setuid** will fail and will set **errno** to:

EPERM if the real-user-ID of the calling-process is not equal to **uid** and its effective-user-ID is not super-user.

EINVAL if **uid** is out of range.

The function **setgid** will fail and will set **errno** to:

EPERM if the real-group-ID of the calling-process is not equal to **gid** and its effective-user-ID is not super-user.

EINVAL if **gid** is out of range.

SEE ALSO

EXEC(BA_OS), GETUID(BA_OS).

LEVEL

Level 1.

NAME

signal — specify what to do upon receipt of a signal

SYNOPSIS

```
#include <signal.h>

void (*signal(sig, func))()
int sig;
void (*func)();
```

DESCRIPTION

The function **signal** allows the calling-process to choose one of three ways in which it is possible to handle the receipt of a specific signal.

The argument **sig** specifies the signal and the argument **func** specifies the choice. The argument **sig** can be assigned any one of the following signals except **SIGKILL**:

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT	quit*
SIGILL	illegal instruction (not reset when caught)*
SIGTRAP	trace trap (not reset when caught)*
SIGABRT	abort*
SIGFPE	floating point exception*
SIGKILL	kill (cannot be caught or ignored)
SIGSYS	bad argument to routine*
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal

* The default action for these signals is an abnormal process termination. See **SIG_DFL**.

SIGUSR1 user-defined signal 1

SIGUSR2 user-defined signal 2

For portability, application-programs should use or catch *only* the signals listed above; other signals are hardware- and implementation-dependent and may have very different meanings or results across systems. (For example, the System V signals **SIGEMT, SIGBUS, SIGSEGV,** and **SIGIOT** are implementation-dependent and are not listed above.) Specific implementations may have other implementation-dependent signals.

The argument **func** is assigned one of three values: **SIG_DFL, SIG_IGN,** or an *address* of a signal-catching function. The argument **func** is declared as type pointer to a function returning **void.** The following actions are prescribed by these values:

SIG_DFL Terminate process upon receipt of a signal.

Upon receipt of the signal **sig,** the receiving process is to be terminated with all of the consequences outlined in EXIT(BA_OS). In addition, if **sig** is one of the signals marked with an asterisk above, implementation-dependent abnormal process termination routines, such as a core dump, may be invoked.

SIG_IGN Ignore signal.

The signal **sig** is to be ignored.

NOTE: The signal **SIGKILL** cannot be ignored.

address Catch signal.

Upon receipt of the signal **sig,** the receiving process is to execute the signal-catching function pointed to by **func.** The signal number **sig** will be passed as the only argument to the signal-catching function. Additional arguments may be passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of **func** for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL** or **SIGTRAP.**

The function **signal** will not catch an invalid function argument, **func,** and results are undefined when an attempt is made to execute the function at the bad address.

Upon return from the signal-catching function, the receiving process will resume execution at the point at which it was interrupted, except for implementation defined signals where this may not be true.

When a signal to be caught occurs during a non-atomic operation such as a call to the READ(BA_OS), WRITE(BA_OS), OPEN(BA_OS), or IOCTL(BA_OS) routine on a slow device (such as a terminal); or occurs during a PAUSE(BA_OS) routine; or occurs during a WAIT(BA_OS) routine that does not return immediately, the signal-catching function will be executed and then the interrupted routine may return a **−1** to the calling-process with **errno** set to **EINTR.**

NOTE: The signal **SIGKILL** cannot be caught.

A call to the function **signal** cancels a pending signal **sig** except for a pending **SIGKILL** signal.

RETURN VALUE

If successful, the function **signal** will return the previous value of the argument **func** for the specified signal **sig**; otherwise, it will return **SIG_ERR** and **errno** will indicate the error.

ERRORS

The function **signal** will fail and will set **errno** to:

EINVAL if **sig** is an illegal signal number or **SIGKILL.**

APPLICATION USAGE

Signals may be sent by the system to an application-program (user-level process) or signals may be sent by one user-level process to another using the KILL(BA_OS) routine. An application-program can catch signals and specify the action to be taken using the SIGNAL(BA_OS) routine. The signals that a portable application-program may *send* are: **SIGKILL, SIGTERM, SIGUSR1**, and **SIGUSR2.**

For portability, application-programs should use only the symbolic names of signals rather than their values and use only the set of signals defined here. Specific implementations may have additional signals.

SEE ALSO

KILL(BA_OS), PAUSE(BA_OS), WAIT(BA_OS), SETJMP(BA_LIB).

FUTURE DIRECTIONS

The end-user level utility KILL(BU_CMD) will be changed to use symbolic signal names rather than numbers.

LEVEL

Level 1.

NAME

sigset, sighold, sigrelse, sigignore — signal management

SYNOPSIS

```
#include <signal.h>

void (*sigset(sig, func))()
int sig;
void (*func)();

int sighold(sig)
int sig;

int sigrelse(sig)
int sig;

int sigignore(sig)
int sig;
```

DESCRIPTION

The functions **sigset**, **sighold**, **sigrelse**, and **sigignore** enhance the signal facility and provide signal management for application processes.

The argument **sig** specifies the signal and the argument **func** specifies the choice. The argument **sig** can be assigned any one of the following signals except **SIGKILL**:

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT	quit*
SIGILL	illegal instruction (not reset when caught)*
SIGTRAP	trace trap (not reset when caught)*
SIGABRT	abort*
SIGFPE	floating point exception*

* The default action for these signals is an abnormal process termination. See **SIG_DFL**.

SIGKILL kill (cannot be caught or ignored)

SIGSYS bad argument to routine*

SIGPIPE write on a pipe with no one to read it

SIGALRM alarm clock

SIGTERM software termination signal

SIGUSR1 user-defined signal 1

SIGUSR2 user-defined signal 2

For portability, application-programs should use or catch *only* the signals listed above; other signals are hardware- and implementation-dependent and may have very different meanings or results across systems. (For example, the System V signals **SIGEMT, SIGBUS, SIGSEGV,** and **SIGIOT** are implementation-dependent and are not listed above.) Specific implementations may have other implementation-dependent signals.

The argument **func** is assigned one of four values: **SIG_DFL, SIG_IGN, SIG_HOLD**, or an *address* of a signal-catching function. The argument **func** is declared as type pointer to a function returning **void**. The following actions are prescribed by these values:

SIG_DFL Terminate process upon receipt of a signal.

Upon receipt of the signal **sig**, the receiving process is to be terminated with all of the consequences outlined in EXIT(BA_OS). In addition, if **sig** is one of the signals marked with an asterisk above, implementation-dependent abnormal process termination routines, such as a core dump, may be invoked.

SIG_IGN Ignore signal.

Any pending signal **sig** is discarded. A pending signal is a signal that has occurred but for which no action has been taken. The system signal action is set to ignore future occurrences of this signal type.

SIG_HOLD Hold signal.

The signal **sig** is to be held. Any pending signal of this type remains held. Only one signal of each type is held.

address Catch signal.

Upon receipt of the signal **sig**, the receiving process is to execute the signal-catching function pointed to by **func**. Any pending signal of this type is released. This address is retained across calls to the other signal management functions, **sighold** and **sigrelse**. The signal number **sig** will be passed as the only argument to the signal-catching function. Before entering the signal-catching function, the value of **func** for the caught signal will be set to **SIG_HOLD**. During normal return from the signal-catching handler, the system signal action is restored to **func** and any held signal of this type is released. If a non-local goto [see SETJMP(BA_LIB)] is taken, the function **sigrelse** must be invoked to restore the system signal action and to release any held signal of this type.

Upon return from the signal-catching function, the receiving process will resume execution at the point at which it was interrupted, except for implementation defined signals where this may not be true.

When a signal to be caught occurs during a non-atomic operation such as a call to the READ(BA_OS), WRITE(BA_OS), OPEN(BA_OS), or IOCTL(BA_OS) routine on a slow device (such as a terminal); or occurs during a PAUSE(BA_OS) routine; or occurs during a WAIT(BA_OS) routine that does not return immediately, the signal-catching function will be executed and then the interrupted routine may return a **−1** to the calling-process with **errno** set to **EINTR**.

The function **sigset** specifies the system signal action to be taken upon receipt of the argument **sig**.

The function **sighold** and the function **sigrelse** establish critical regions of code. A call to the function **sighold** is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by the function **sigrelse**. The function **sigrelse** restores the system signal action to the action that was previously specified by the function **sigset**.

The function **sigignore** sets the action for the argument **sig** to **SIG_IGN**.

RETURN VALUE

If successful, the function **sigset** will return the previous value of the system signal action for the specified signal **sig**; otherwise, it will return **SIG_ERR** and **errno** will indicate the error.

For the functions **sighold**, **sigrelse**, and **sigignore** a value of **0** will be returned upon success. Otherwise, a value of **−1** will be returned and **errno** will indicate the error.

ERRORS

Under the following conditions, the functions **sigset**, **sighold**, **sigrelse**, and **sigignore** will fail and will set **errno** to:

EINVAL if **sig** is an illegal signal number or **SIGKILL** or if the default handling of **sig** cannot be changed.

APPLICATION USAGE

The functions **sigset**, **sighold**, **sigrelse**, and **sigignore** were added in System V Release 3.0.

For portability, application-programs should use only the symbolic names of signals rather than their values and use only the set of signals defined here. Specific implementations may have additional signals.

The other signal management routine, SIGNAL(BA_OS), should not be used in conjunction with these routines for a particular signal type.

SEE ALSO

KILL(BA_OS), PAUSE(BA_OS), SIGNAL(BA_OS), WAIT(BA_OS), SETJMP(BA_LIB).

LEVEL

Level 1.

NAME

time — get time

SYNOPSIS

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

time_t time(tloc)
time_t *tloc;
```

DESCRIPTION

The function **time** returns the value of time in seconds since 00:00:00 GMT, 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

If successful, the function **time** will return the value of time; otherwise, it will return **−1**.

SEE ALSO

STIME(BA_OS).

LEVEL

Level 1.

NAME

ustat — get file system statistics

SYNOPSIS

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

int ustat(dev, buf)
dev_t dev;
struct ustat *buf;

DESCRIPTION

The function **ustat** returns information about a mounted file system.

The argument **dev** is a device number identifying a device containing a mounted file-system. The value of **dev** is obtained from the field **st_dev** of the structure **stat** [see STAT(BA_OS)].

The argument **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 i-nodes */
char    f_fname[6]; /* file-system name or null */
char    f_fpack[6]; /* file-system pack or null */
```

The last two fields, **f_fname** and **f_fpack** may not have significant information on all systems, and, in that case, will contain the null character.

RETURN VALUE

If successful, the function **ustat** will return **0**; otherwise, it will return **–1** and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **ustat** will fail and will set **errno** to:

EINVAL if **dev** is not the device number of a device containing a mounted file-system.

SEE ALSO

STAT(BA_OS).

LEVEL

Level 1.

NAME

write — write on a file

SYNOPSIS

int write(fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

The function **write** attempts to write **nbyte** bytes from the buffer pointed to by the argument **buf** to the file associated with the argument **fildes**.

The argument **fildes** is an open file-descriptor [see **file-descriptor** in **Chapter 4 — Definitions**].

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file-pointer associated with the argument **fildes**. Upon returning from the function **write**, the file-pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, such as a terminal, writing always takes place starting at the current position. The value of a file-pointer associated with such a device is undefined [see OPEN(BA_OS)].

If the **O_APPEND** flag of the file status flags is set, the file-pointer will be set to the end of the file prior to each **write** operation.

For ordinary files, if the **O_SYNC** flag of the file status flags is set, the write should not return until both the file data and file status have been physically updated. For block special files, if **O_SYNC** is set, the write should not return until the data has been physically updated. The way the data reaches the physical media is implementation- and hardware-dependent.

When attempting to write to an ordinary file with enforced record locking enabled [see CHMOD(BA_OS)], and all or part of the file to be written has a read or write lock owned by another process (i.e., a blocking lock):

> If **O_NDELAY** is set, the function **write** will return **−1** and **errno** will be set to **EAGAIN**.
>
> If **O_NDELAY** is clear, the function **write** will sleep until all blocking locks are removed, or the function **write** is terminated by a signal.

If a **write** requests that more bytes be written than there is room for (e.g., beyond the user process' file size limit [see ULIMIT(BA_OS)] or the physical end of a medium), 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 will return 20-bytes. The next **write** of a non-zero number of bytes will give a failure return (except as noted for pipes and FIFOs below).

If a **write** to a pipe (or FIFO) of {PIPE_BUF} bytes or less is requested and less than **nbytes** bytes of free space is available in the pipe, one of the following will occur:

> If the **O_NDELAY** flag is clear, the process will block until at least **nbytes** of space is available in the pipe and then the **write** will take place, or
>
> If the **O_NDELAY** flag is set, the process will not block and the function **write** will return **0**.

If a **write** to a pipe (or FIFO) of more than {PIPE_BUF} bytes is requested, one of the following will occur:

> If the **O_NDELAY** flag is clear, the process will block if the pipe is full. As space becomes available in the pipe, the data from the **write** request will be written piecemeal — in multiple smaller amounts until the request is fulfilled. Thus, data from a **write** request of more than {PIPE_BUF} bytes may be interleaved on arbitrary byte boundaries with data written by other processes.
>
> If the **O_NDELAY** flag is set and the pipe is full, the process will not block and the function **write** will return **0**.
>
> If the **O_NDELAY** flag is set and the pipe is not full, the process will not block and as much data as will currently fit in the pipe will be written, and the function **write** will return the number of bytes written. In this case, only part of the data are written, but what data are written will not be interleaved with data from other processes.

In contrast to **write** requests of more than {PIPE_BUF} bytes, data from a **write** request of {PIPE_BUF} bytes or less will never be interleaved in the pipe with data from other processes.

RETURN VALUE

If successful, the function **write** will return the number of bytes actually written; otherwise, it will return **−1**, the file-pointer will remain unchanged, and **errno** will indicate the error.

ERRORS

Under the following conditions, the function **write** will fail and will set **errno** to:

EBADF	if **fildes** is not a valid file descriptor open for writing.
EPIPE and **SIGPIPE** signal	if an attempt is made to write to a pipe that is not open for reading by any process.
EFBIG	if an attempt was made to write a file that exceeds the process' file size limit or the system's maximum file size [see ULIMIT(BA_OS)].
EINTR	if a signal was caught during the **write** operation.
ENOSPC	if there is no free space remaining on the device containing the file.
EIO	if a physical I/O error has occurred.
ENXIO	if the device associated with the file-descriptor is a block-special or character-special file and the file-pointer value is out of range.
EAGAIN	if enforced record locking was enabled, **O_NDELAY** was set and there were record-locks on the file.
ENOLCK	if enforced record locking was enabled and {LOCK_MAX} regions are already locked in the system.
EDEADLK	if enforced record locking was enabled, **O_NDELAY** was clear and a deadlock condition was detected.

APPLICATION USAGE

Normally, applications should use the *stdio* routines to open, close, read and write files. Thus, if an application had used the FOPEN(BA_OS) *stdio* routine to open a file, it would use the FWRITE(BA_OS) *stdio* routine rather than the WRITE(BA_OS) routine to write it.

Because they are not atomic, **write** requests of **nbytes** greater than {PIPE_BUF} bytes to a pipe (or FIFO) should only be used when just two cooperating processes, one reader and one writer, are using a pipe.

When **O_NDELAY** is set, portable application-programs should test for two conditions to determine that no data is currently available, for example:

```
fildes = open(path, O_WRONLY | O_NDELAY);
ret = write(fildes, buf, nbyte);
if (ret == 0 || (ret == -1 && errno == EAGAIN)) {
        .
        .  /* Data not available now. */
        .
}
```

Use of the O_SYNC flag should be used by applications that require extra reliability at the cost of performance.

SEE ALSO

CREAT(BA_OS), DUP(BA_OS), LSEEK(BA_OS), OPEN(BA_OS), PIPE(BA_OS), ULIMIT(BA_OS).

LEVEL

Level 1.

Chapter 7
General Library Routines

NAME

clock — report CPU time used

SYNOPSIS

long clock()

DESCRIPTION

The function **clock** returns the amount of CPU time (in microseconds) 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(BA_OS), PCLOSE(BA_OS), or SYSTEM(BA_OS) routine.

APPLICATION 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

ctime, localtime, gmtime, asctime, tzset — convert date and time to string

SYNOPSIS

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

char *ctime(clock)
time_t *clock;

struct tm *localtime(clock)
time_t *clock;

struct tm *gmtime(clock)
time_t *clock;

char *asctime(tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset()
```

DESCRIPTION

The function **ctime** converts a value of type **time_t**, pointed to by **clock**, representing the time in seconds since 00:00:00 GMT, January 1, 1970 [see TIME(BA_OS)] and returns a pointer to a 26-character string in the following form:

```
Sun Sep 16 01:03:52 1973
```

All the fields have constant width.

The functions **localtime** and **gmtime** return pointers to the structure **tm**, described below:

> The function **localtime** corrects for the time-zone and possible Daylight Saving Time.
>
> The function **gmtime** converts directly to Greenwich Mean Time (GMT), which is the time the system uses.

The function **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, the external variables and the **tm** structure are in the **<time.h>** header file. The structure **tm** includes the following members:

```
int tm_sec;    /* number of seconds past */
               /* the minute (0-59) */
int tm_min;    /* number of minutes past */
               /* the hour (0-59) */
int tm_hour;   /* current hour (0-23) */
int tm_mday;   /* day of month (1-31) */
int tm_mon;    /* month of year (0-11) */
int tm_year;   /* current year -1900 */
int tm_wday;   /* day of week (Sunday=0) */
int tm_yday;   /* day of year (0-365) */
int tm_isdst;  /* daylight savings time flag */
```

The value of **tm_isdst** is non-zero if Daylight Saving Time is in effect.

The external **long** variable **timezone** contains the difference, in seconds, between GMT and local standard time (in EST, **timezone** is **5*60*60)**; the external variable **daylight** is non-zero only if the standard USA Daylight Saving Time conversion should be applied. The program compensates for the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named **TZ** is present, **asctime** uses the contents of the variable to override the default time-zone. The value of **TZ** must be a three-letter time-zone name, followed by an optional minus sign (for zones east of Greenwich) and a series of digits representing the difference between local time and Greenwich Mean Time in hours; this is followed by an optional three-letter name for a daylight time-zone. For example, the setting for New Jersey would be **EST5EDT**. The effects of setting **TZ** are thus to change the values of the external variables **timezone** and **daylight**. In addition, the time-zone names contained in the external variable

```
char *tzname[2] = { "EST", "EDT" };
```

are set from the environment variable **TZ**. The function **tzset** sets these external variables from **TZ**; the function **tzset** is called by **asctime** and may also be called explicitly by the user.

APPLICATION USAGE

The return values point to static data whose content is overwritten by each call.

SEE ALSO

TIME(BA_OS), GETENV(BA_LIB).

FUTURE DIRECTIONS

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

LEVEL

Level 1.

NAME

floor, ceil, fmod, fabs — floor, ceiling, remainder, absolute value functions

SYNOPSIS

#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fmod(x, y)
double x, y;

double fabs(x)
double x;

DESCRIPTION

The function **floor** returns the largest integer (as a double-precision number) not greater than **x**.

The function **ceil** returns the smallest integer not less than **x** .

The function **fmod** returns the floating-point remainder of the division of **x** by **y**, **x** if **y** is zero or if **x/y** would overflow. Otherwise the number is f with the same sign as **x**, such that $\mathbf{x} = i\mathbf{y} + f$ for some integer i, and $|f| < |y|$.

The function **fabs** returns the absolute value of **x**, i.e., $|\mathbf{x}|$.

SEE ALSO

ABS(BA_LIB).

LEVEL

Level 1.

NAME

getopt — get option letter from argument vector

SYNOPSIS

int getopt(argc, argv, optstring)
int argc;
char *argv[ ], *optstring;

extern char *optarg;
extern int optind, opterr;

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 must 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 must 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.

The function **getopt** is the command-line parser that will enforce the rules of this command syntax standard.

RETURN VALUE

The function **getopt** prints an error message on **stderr** and returns a question-mark (**?**) when it encounters an option letter not included in **opt-string**. Setting **opterr** to a **0** will disable this error message.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b** and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char *argv [ ];
{
    int c;
    int bflg, aflg, errflg;
    char *ifile;
    char *ofile;
    extern char *optarg;
    extern int optind;
    . . .
    while ((c = getopt(argc, argv, "abf:o:")) != EOF)
        switch (c) {
        case 'a': if (bflg)
                      errflg++;
                  else
                      aflg++;
                  break;
        case 'b': if (aflg)
                      errflg++;
                  else
                      bproc( );
                  break;
        case 'f': ifile = optarg;
                  break;
        case 'o': ofile = optarg;
                  break;
        case '?': errflg++;
        }
    if (errflg) {
        fprintf(stderr, "usage: . . . ");
        exit(2);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], 4)) {
    . . .
}
```

LEVEL

Level 1.

NAME

printf, fprintf, sprintf — print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf(format [ , arg ] ...)
char *format;

int fprintf(stream, format [ , arg ] ...)
FILE *stream;
char *format;

int sprintf(s, format [ , arg ] ...)
char *s, *format;
```

DESCRIPTION

The function **printf** places output on the standard output stream **stdout.**

The function **fprintf** places output on the named output **stream.**

The function **sprintf** places output, followed by the null character (**0**) in consecutive bytes starting at ***s.** It is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **0** in the case of **sprintf**) or a negative value if an output error was encountered.

Each of these functions converts, formats and prints its **args** under control of the **format.** The **format** is a character-string that contains three types of objects defined below:

1. plain-characters that are simply copied to the output stream;

2. escape-sequences that represent non-graphic characters; and

3. conversion-specifications.

The following escape-sequences produce the associated action on display devices capable of the action:

b Backspace.
Moves the printing position to one character before the current position, unless the current position is the start of a line.

f Form Feed.
Moves the printing position to the initial printing position of the next logical page.

\n New line.
Moves the printing position to the start of the next line.

\r Carriage return.
Moves the printing position to the start of the current line.

\t Horizontal tab.
Moves the printing position to the next implementation-defined horizontal tab position on the current line.

\v Vertical tab.
Moves the printing position to the start of the next implementation-defined vertical tab position.

Each conversion specification is introduced by the character %. After the character %, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional string of decimal digits to specify a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (–), described below, has been given) to the field width.

A *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions (the field is padded with leading zeros), the number of digits to appear after the decimal point for the **e**, **E** and **f** conversions, the maximum number of significant digits for the **g** and **G** conversion; or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (**.**) followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.

An optional **l** (ell) to specify that a following **d**, **i**, **o**, **u**, **x** or **X** conversion character applies to a long integer **arg**. An **l** before any other conversion character is ignored.

A conversion character (see below) that indicates the type of conversion to be applied.

A *field width* or *precision* may be indicated by an asterisk (*) instead of a digit string. In this case, an integer **arg** supplies the field width or precision. The **arg** that is actually converted is not fetched until the conversion letter is seen, so the **arg**s specifying field width or precision must appear **before** the **arg** (if any) to be converted. If the *precision* argument is negative, it will be changed to zero.

The *flag* characters and their meanings are:

– The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or -).

blank If the first character of a signed conversion is not a sign, a blank will be prepended to the result. This means that if the blank and + flags both appear, the blank flag will be ignored.

The value is to be converted to an *alternate form*. For **c**, **d**, **i**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversion, a non-zero result will have **0x** or **0X** prepended to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result as they normally are.

Each conversion character results in fetching zero or more **args**. 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.

The conversion characters and their meanings are:

d,i,o,u,x,X The integer **arg** is converted to signed decimal (**d** or **i**), **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* component of **arg** specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it will be expanded with leading zeroes. The default precision is **1**. The result of converting a zero value with a precision of **0** is a null string.

f The float or double **arg** is converted to decimal notation in the style **[–]ddd.ddd**, where the number of digits after the decimal point is equal to the *precision* specification. If the *precision* is omitted from **arg**, six digits are output; if the *precision* is explicitly **0**, no decimal point appears.

e,E The float or double **arg** is converted to the style **[−]d.ddde±dd**, where there is one digit before the decimal point and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is **0**, no decimal point appears. The **E** conversion character will produce a number with **E** instead of **e** introducing the exponent.

The exponent always contains at least two digits. However, if the value to be printed is greater than or equal to **1E+100**, additional exponent digits will be printed as necessary.

g,G The float or double **arg** is printed in style **f** or **e** (or in style **E** in the case of a **G** conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than **−4** or greater than the precision. Trailing zeroes are removed from the result. A decimal point appears only if it is followed by a digit.

c The character **arg** is printed.

s The **arg** is taken to be a string (character pointer) and characters from the string are printed until a null character (**0**) is encountered or the number of characters indicated by the *precision* specification of **arg** is reached. If the precision is omitted from **arg**, it is taken to be infinite, so all characters up to the first null character are printed. A **NULL** value for **arg** will yield undefined results.

% Print a %; no argument is converted.

If the character after the % is not a valid conversion character, the results of the conversion are undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by **printf** and **fprintf** are printed as if the PUTC(BA_LIB) routine had been called.

RETURN VALUE

The functions **printf**, **fprintf**, and **sprintf** return the number of characters transmitted, or return **−1** if an error was encountered.

EXAMPLE

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));
```

SEE ALSO

PUTC(BA_LIB), SCANF(BA_LIB), FOPEN(BA_OS).

FUTURE DIRECTIONS

The function **printf** will make available character string representations for ∞ and "not a number" (NaN: a symbolic entity encoded in floating point format) to support the **IEEE P754** standard.

LEVEL

Level 1.

NAME

scanf, fscanf, sscanf — convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf(format [ , pointer ] ...)
char *format;

int fscanf(stream, format [ , pointer ] ...))
FILE *stream;
char *format;

int sscanf(s, format [ , pointer ] ...)
char *s, *format;
```

DESCRIPTION

The function **scanf** reads from the standard input stream **stdin**.

The function **fscanf** reads from the named input **stream**.

The function **sscanf** reads from the character string **s**.

Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string **format** described below and a set of **pointer** arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.

2. An ordinary character (not %), which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing the character *****, a decimal digit string that specifies an optional numerical maximum field width, an optional letter **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument unless assignment suppression was indicated by the character *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the maximum field width, if one is specified, is exhausted. For all descriptors except the character [and the character **c**, white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

% a single % is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

i an integer is expected; the corresponding argument should be an integer pointer. The value of the next input item, interpreted according to C conventions, will be stored; a leading **0** implies octal, a leading **0x** implies hexadecimal; otherwise, decimal is assumed.

n causes the total number of characters (including white space) that have been scanned so far since the function call to be stored; the corresponding argument should be an integer pointer. No input is consumed.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a **float**. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point; followed by an optional exponent field consisting of an **E** or an **e**, followed by an optionally signed integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters called the *scanset* and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string.

There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus **[0123456789]** may be expressed **[0–9]**. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The character – will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically. At least one character must match for this conversion to be considered successful.

If an invalid conversion character follows the %, the results of the operation may not be predictable.

The conversion characters **d**, **u**, **o**, **x**, and **i** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

The **scanf** conversion terminates at end of file, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

RETURN VALUE

These routines return the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

APPLICATION USAGE

Trailing white space (including a new-line) is left unread unless matched in the control string.

The success of literal matches and suppressed assignments is not directly determinable.

EXAMPLE

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 string **56\0** in **name**. The next call to **getchar** [see GETC(BA_LIB)] will return **a**.

SEE ALSO

GETC(BA_LIB), PRINTF(BA_LIB), STRTOD(BA_LIB), STRTOL(BA_LIB).

FUTURE DIRECTIONS

The function **scanf** will make available character string representations for ∞ and "not a number" (NaN: a symbolic entity encoded in floating point format) to support the **IEEE P754** standard.

LEVEL

Level 1.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strdup, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok — string operations

SYNOPSIS

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

char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;
size_t n;

int strcmp(s1, s2)
char *s1, *s2;

int strncmp(s1, s2, n)
char *s1, *s2;
size_t n;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;
size_t n;

char *strdup(s1)
char *s1;

int strlen(s)
char *s;

char *strchr(s, c)
char *s;
int c;

char *strrchr(s, c)
char *s;
int c;

char *strpbrk(s1, s2)
char *s1, *s2;

int strspn(s1, s2)
char *s1, *s2;

int strcspn(s1, s2)
char *s1, *s2;

char *strtok(s1, s2)
char *s1, *s2;
```

DESCRIPTION

The arguments **s1**, **s2**, and **s** point to strings (arrays of characters terminated by a null character). The functions **strcat**, **strncat**, **strcpy**, **strncpy**, and **strtok** all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**. The type **size_t** is defined in the **<sys/types.h>** header file.

The function **strcat** appends a copy of string **s2** to the end of string **s1**.

The function **strncat** appends at most **n** characters. Each returns a pointer to the null-terminated result.

The function **strcmp** compares its arguments 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**.

The function **strncmp** makes the same comparison but looks at at most **n** characters.

The function **strcpy** copies string **s2** to **s1**, stopping after the null character has been copied.

The function **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**.

The function **strdup** returns a pointer to a new string, which is a duplicate of the string pointed to by **s1**. Space for the new string is obtained using MALLOC(BA_OS). A **NULL** pointer is returned if the new string cannot be created.

The function **strlen** returns the number of characters in **s**, not including the terminating null character.

The function **strchr** or the function **strrchr** returns a pointer to the first (last) occurrence of character **c** 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.

The function **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**.

The function **strspn** or the function **strcspn** returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

The function **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**, returning a pointer to the first character of each subsequent token. A null character will have been written into **s1** by **strtok** immediately following the token. The separator string **s2** may be different from call to call. When no token remains in **s1**, a **NULL** pointer is returned.

APPLICATION USAGE

All these functions are declared by the <**string.h**> header file.

Both **strcmp** and **strncmp** use native character comparison. The sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

SEE ALSO

MEMORY(BA_LIB).

FUTURE DIRECTIONS

The type of value returned by **strlen** will be declared as **size_t**.

LEVEL

Level 1.

Appendix
Changes From Issue 2 Volume 1

Appendix
Changes From Issue 2 Volume 1

This Appendix documents changes from Issue 2, Volume 1 of the *Base System Definition* of the *System V Interface Definition.* Only changes that serve to clarify information, provide additional information, or identify incorrect information that appears in Issue 2, Volume 1 are documented. Changes that do not alter meaning are not listed below.

The paragraphs below identify specific changes to detailed component definitions:

1. BASE SYSTEM DIFFERENCES

Environment.

ERRORS(BA_ENV) The errno **ENOENT** has the added meaning that a path-name argument to a BA_OS function is longer than {PATH_MAX} characters. This affects the functions ACCESS(BA_OS), CHDIR(BA_OS), CHMOD(BA_OS), CHOWN(BA_OS), CREAT(BA_OS), EXEC(BA_OS), FOPEN(BA_OS), LINK(BA_OS), MKNOD(BA_OS), OPEN(BA_OS), STAT(BA_OS), UNLINK(BA_OS), and UTIME(BA_OS).

FUTURE DIRECTION: To conform with the IEEE POSIX standard, when it is adopted as a full-use standard, the value of **errno** indicating that a path-name argument exceeds {PATH_MAX} characters may be changed. Currently, the POSIX trial-use draft specifies **ENAMETOOLONG** for this condition.

TERMIO(BA_ENV) Issue 2 incorrectly listed B38400 as a valid baud rate. This baud rate should be deleted.

OS Service Routines.

CHMOD(BA_OS) Issue 2 identified the access permission bit **01000** as (execute or search by group) in the **FUTURE DIRECTIONS** section; this should correctly refer to access permission bit **00010**.

FERROR(BA_OS) In Issue 2, the first paragraph under **DESCRIPTION** should be changed to the following:

> The function **ferror** determines if an I/O error (e.g., EINTR, ENOSPC) has occurred when reading from or writing to the file associated with the named **stream**.

FOPEN(BA_OS) The last statement in the **RETURN VALUE** section should state:

> The function **fopen** or the function **fdopen** may also fail if there are no free *stdio* streams and may not set **errno**.

MALLOC(BA_OS) Issue 2 defined the default value for **maxfast** to be **0**. This value should be defined as implementation dependent.

OPEN(BA_OS) Issue 2 specified that the **mode** argument to the OPEN(BA_OS) routine was required. This argument should be defined to be optional, i.e., **int open (path, oflag [,mode])**.

General Library Routines

REGEXP(BA_LIB) Issue 2 incorrectly specified the function names on the NAME line. This line should state:

> compile, step, advance — regular-expression compile and match routines

SETJMP(BA_LIB) In Issue 2, the last paragraph under **APPLICATION USAGE** should be deleted.

The last sentence of the last paragraph in the **DESCRIPTION** section should be changed to the following:

> 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.

TMPFILE(BA_LIB) Issue 2 incorrectly described the **RETURN VALUE**, which should correctly read:

> If the temporary file cannot be opened, a NULL pointer is returned.

VPRINTF(BA_LIB) Issue 2 presented an incomplete statement in the EXAMPLE. The line

> **(void) fprintf(stderr, " ERR** in %s:

should correctly read

> **(void) fprintf(stderr, " ERR** in %s: " ,
> **va_arg (args, char *));**

The following section should appear under the **APPLICATION USAGE** section.

> Specification of a second argument to the **va_arg** macro of type **char, short**, or **float** is non-portable, since arguments seen by the called function are not **char, short**, or **float**. The C compiler converts **char** and **short** arguments to **int** and converts **float** arguments to double before passing them to a function.

2. KERNEL EXTENSION DIFFERENCES

MSGOP(KE_OS) Issue 2 incorrectly specified the function names on the NAME line. This line should state:

> msgsnd, msgrcv — message operations

PROFIL(KE_OS) Issue 2 incorrectly specified types for two of the arguments to **profil**. The argument **buff** should be defined as **short *buff** and the argument **offset** should be defined as **void (*offset)()**.

PTRACE(KE_OS) The description of the action to be taken when the value of the argument **request** is 7 should read as follows:

7 This request causes the child to resume execution. The **data** argument is taken as a signal number and the child's execution continues at location **addr** as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be

ignored, or the value of the signal that caused the stop. If **addr** is 1, then execution continues from where it stopped.

Upon successful completion, the value of **data** is returned to the parent. This request will fail 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**.

SEMGET(KE_OS)

Issue 2 incorrectly included a paragraph in the **DESCRIPTION** section. The last paragraph, which states "The data structure associated with each semaphor in the set is not initialized. The function **semctl** with the command **SETVAL** or **SETALL** can be used to initialize each semaphore.", should be removed.

SEMOP(KE_OS)

Issue 2 incorrectly specified use of **IPC_CREAT** instead of **IPC_NOWAIT**. All instances of **IPC_CREAT** should be replaced by **IPC_NOWAIT**.

SHMOP(KE_OS)

Issue 2 incorrectly specified the function names on the NAME line. This line should state:

shmat, shmdt — shared-memory-operations

Part III

Terminal Interface Extension Definition

Chapter 8
Introduction

8.1 OVERVIEW

The Terminal Interface Extension (TI) consists of the facilities provided by the *curses/terminfo* package to allow application programs to perform terminal-handling functions in a way that is independent of the type of the terminal actually in use. Currently, the *curses/terminfo* package supports asynchronous character terminals.

The System V Base, the Basic Utilities Extension, the Advanced Utilities Extension, and the Software Development Extension are prerequisites for the Terminal Interface Extension.

The components of the Terminal Interface Extension are new in System V Release 2.

8.2 DESCRIPTION

DATA FILES

/usr/lib/terminfo/?/*

UTILITIES

tic **tput**

LIBRARY ROUTINES

General Routines

addch	has_ic	nl	standend
addstr	has_il	nocbreak	standout
attroff	idlok	nodelay	subpad††
attron	inch	noecho	subwin
attrset	initscr	nonl	touchline††
baudrate	insch	noraw	touchwin
beep	insertln	overlay	typeahead
box	intrflush	overwrite	unctrl
cbreak	keyname††	pechochar††	ungetch
clear	keypad	pnoutrefresh	waddch
clearok	killchar	prefresh	waddstr
clrtobot	leaveok	printw	wattroff
clrtoeol	longname	raw	wattron
copywin††	move	refresh	wattrset
def_prog_mode	mvaddch	reset_prog_mode	wclear
def_shell_mode	mvaddstr	reset_shell_mode	wclrtobot
delay_output	mvdelch	resetterm **	wclrtoeol
delch	mvgetch	resetty	wdelch
deleteln	mvgetstr	saveterm **	wdeleteln
delwin	mvinch	savetty	wechochar††
doupdate	mvinsch	scanw	werase
echo	mvprintw	scr_dump††	wgetch
echochar††	mvscanw	scr_init††	wgetstr
endwin	mvwaddch	scr_restore††	winch
erase	mvwaddstr	scroll	winsch
erasechar	mvwdelch	scrollok	winsertln
fixterm **	mvwgetch	set_term	wmove
flash	mvwgetstr	setscrreg	wnoutrefresh
flushinp	mvwin	slk_clear††	wprintw
getbegyx††	mvwinch	slk_init††	wrefresh
getch	mvwinsch	slk_label††	wscanw
getmaxyx††	mvwprintw	slk_noutrefresh††	wsetscrreg
getstr	mvwscanw	slk_refresh††	wstandend
gettmode **	newpad	slk_restore††	wstandout
getyx	newterm	slk_set††	
halfdelay††	newwin	slk_touch††	

Terminfo Level Routines

mvcur	**setupterm**	**tigetstr**	**vidattr**
putp	**tigetflag**	**tparm**	**vidputs**
setterm **	**tigetnum**	**tputs**	

Termcap Compatibility Routines

tgetent **	**tgetnum** **	**tgoto** **	**tputs**
tgetflag **	**tgetstr** **		

** Level 2: December 1, 1985.

8.3 DEFINITIONS

The following environment variables are used by the components of the TI extension. See SH(BU_CMD) for information on the shell environment.

TERM
The environmental variable **TERM** usually contains a user's current terminal type and can be set by the user.

TERMINFO
The environmental variable **TERMINFO**, if set, contains the place where local terminal descriptions can be found. **TERMINFO** can be set by the user. If it is set, any program using CURSES(TI_LIB) will check the **TERMINFO** location for a terminal's description before checking **/usr/lib/terminfo**, the standard location for terminal descriptions. See CURSES(TI_LIB) and TERMINFO(TI_ENV) for further information.

LINES and **COLUMNS**
The environmental variables **LINES** and **COLUMNS**, if set, contain the number of lines and number of columns, respectively, on a terminal screen and can be set by the user. If defined, the values of these variables, **LINES** and **COLUMNS**, will override the screen size values given in a terminal's *terminfo* [see TERMINFO(TI_ENV)] description. See CURSES(TI_LIB) for further information.

8.4 TRADEMARKS

Tektronix is a registered trademark of Tektronix, Inc.
TeleVideo is a registered trademark of TeleVideo Systems, Inc.
VT100 is a trademark of Digital Equipment Corporation.
LSI is a trademark of Lear Siegler, Inc.
HP is a trademark of Hewlett-Packard Co.
Tektronix 4010 is a registered trademark of Tektronix, Inc.
Beehive is a trademark of Beehive International.
Ann Arbor is a trademark of Ann Arbor Terminals, Inc.
Teleray is a trademark of Research, Inc.
Micro-Term, ACT and MIME are trademarks of Micro-Term, Inc.
Concept is a trademark of Human Designed Systems, Inc.

Chapter 9
Environment

NAME

terminfo – terminal capability database

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

The *terminfo* database describes terminals, by giving a set of capabilities which they have, by describing how operations are performed, by describing padding requirements, and by specifying initialization sequences. The *terminfo* database is built by using the TIC(TI_CMD) compiler.

The *terminfo* source files consist of entries that contain a number of comma-separated fields. White space after each comma is ignored. The first entry for each terminal gives the names which are known for the terminal, separated by vertical bar (|) characters. The first name given is the most common abbreviation for the terminal; the last name given should be a long name fully identifying the terminal; and all others are understood as synonyms for the terminal name. All names but the last should be in lowercase letters and contain no blanks; the last name may well contain uppercase letters and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, "att4424". Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. The following suffixes should be used where possible:

Suffix	Meaning
-w	Wide mode (more than 80 columns)
-am	With automatic margins (usually default)
-nam	Without automatic margins
-n	Number of lines on the screen (e.g., -60)
-na	No arrow keys (leave them in local)
-np	Number of pages of memory (e.g., -8p)
-rv	Reverse video

To avoid conflicts with the naming conventions used in describing the different modes of a terminal (e.g., **-w**), it is recommended that a terminal's root name not contain hyphens. Further, it is good practice to make all terminal names used in the *terminfo* database unique.

Capabilities

In the table below, "**Variable**" is the name by which the programmer (at the *terminfo* level) accesses the capability. "**Capname**" is the short name used in the text of the database, and is used by a person updating the database. The "**Termcap Code**" is the two letter code that corresponds to the old *termcap* capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short. Whenever possible, names are chosen to be the same as or similar to the **ANSI** X3.64-1979 standard. Semantics are also intended to match those of the specification.

All string capabilities listed below may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section of the table below, are denoted by the string **key_** at the beginning of their variable name. The following indicators may appear at the end of the **Description** for a variable.

(G) indicates that the string is passed through **tparm**( ) with parms as given (#*i*).

(*) indicates that padding may be based on the number of lines affected.

($\#_i$) indicates the i^{th} parameter.

Variable	Cap-name	Term-cap	Description Code
Booleans:			
auto_left_margin	bw	bw	**cub1** wraps from col 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting
dest_tabs_magic_smso	xt	xt	Destructive tabs, magic **smso** char
eat_newline_glitch	xenl	xn	Newline ignored after 80 cols
erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type (e.g., dialup, switch)
hard_copy	hc	hc	Hardcopy terminal
hard_cursor	chts	HC	Cursor is hard to see
has_meta_key	km	km	(Reserved)
has_status_line	hs	hs	Has extra "status line"
insert_null_glitch	in	in	Insert mode distinguishes nulls
memory_above	da	da	Display may be retained above screen

Variable	Cap-name	Term-cap	Description Code
memory_below	db	db	Display may be retained below screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
needs_xon_xoff	nxon	nx	Padding won't work, xon/xoff required
(Reserved)	xsb	xb	
no_pad_char	npc	NP	Pad character doesn't exist.
non_rev_rmcup	nrrmc	NR	**smcup** does not reverse **rmcup**.
over_strike	os	os	Terminal overstrikes
prtr_silent	mc5i	5i	Printer won't echo on screen.
status_line_esc_ok	eslok	es	Escape can be used on the status line
tilde_glitch	hz	hz	Cannot print tildes
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns	cols	co	Number of columns in a line
init_tabs	it	it	Tabs initially every # spaces
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of cols in each label
lines	lines	li	Number of lines on screen or page
lines_of_memory	lm	lm	Lines of memory if > **lines**, 0=varies
magic_cookie_glitch	xmc	sg	Number of blank chars left by **smso** or **rmso**
num_labels	nlab	Nl	Number of labels on screen (start at 1)
padding_baud_rate	pb	pb	Lowest baud where padding is needed
virtual_terminal	vt	vt	(Reserved)
width_status_line	wsl	ws	Number of columns in status line
Strings:			
acs_chars	acsc	ac	Graphic charset pairs aAbBcC - default=vt100
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
carriage_return	cr	cr	Carriage return (*)
change_scroll_region	csr	cs	Change to lines #1 through #2 (G)
char_padding	rmp	rP	Like **ip** but when in replace mode
clear_all_tabs	tbc	ct	Clear all tab stops
clear_margins	mgc	MC	Clear left and right soft margins
clear_screen	clear	cl	Clear screen and home cursor (*)
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display (*)
column_address	hpa	ch	Horizontal position absolute (G)
command_character	cmdch	CC	(Reserved)
cursor_address	cup	cm	Cursor motion to row #1 col #2 (G)

Variable	Cap-name	Term-cap	Description Code
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no **cup**)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move cursor left one space
cursor_mem_address	mrcup	CM	Memory relative cursor addressing
cursor_normal	cnorm	ve	Make cursor appear normal (undo **vs/vi**)
cursor_right	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll	ll	ll	Last line, first column (if no **cup**)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
delete_character	dch1	dc	Delete character (*)
delete_line	dl1	dl	Delete line (*)
dis_status_line	dsl	ds	Disable status line
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	Enable alternate char set
enter_alt_charset_mode	smacs	as	Start alternate character set
enter_am_mode	smam	SA	Turn on automatic margins
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use **cup**
enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode
enter_insert_mode	smir	im	Insert mode (enter)
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_secure_mode	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode	smso	so	Begin standout mode
enter_underline_mode	smul	us	Start underscore mode
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking
erase_chars	ech	ec	Erase #1 characters (G)
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_am_mode	rmam	RA	Turn off automatic margins
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use **cup**
exit_delete_mode	rmdc	ed	End delete mode
exit_insert_mode	rmir	ei	End insert mode
exit_standout_mode	rmso	se	End standout mode
exit_underline_mode	rmul	ue	End underscore mode
exit_xon_mode	rmxon	RX	Turn off xon/xoff handshaking
flash_screen	flash	vb	Visible bell (may not move cursor)
form_feed	ff	ff	Hardcopy terminal page eject (*)
from_status_line	fsl	fs	Return from status line

Variable	Cap-name	Term-cap	Description Code
init_1string	is1	i1	Terminal initialization string
init_2string	is2	is	Terminal initialization string
init_3string	is3	i3	Terminal initialization string
init_file	if	if	Name of file containing **is**
init_prog	iprog	iP	Path name of program for init
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line (*)
insert_padding	ip	ip	Insert pad after character inserted(*)
key_a1	ka1	K1	KEY_A1, Upper left of keypad
key_a3	ka3	K3	KEY_A3, Upper right of keypad
key_b2	kb2	K2	KEY_B2, Center of keypad
key_c1	kc1	K4	KEY_C1, Lower left of keypad
key_c3	kc3	K5	KEY_C3, Lower right of keypad
key_backspace	kbs	kb	KEY_BACKSPACE, Sent by backspace key
key_beg	kbeg	@1	KEY_BEG, Beg(inning) key
key_btab	kcbt	kB	KEY_BTAB, Back tab key
key_cancel	kcan	@2	KEY_CANCEL, Cancel key
key_catab	ktbc	ka	KEY_CATAB, Sent by clear-all-tabs key
key_clear	kclr	kC	KEY_CLEAR, Sent by clear screen or erase key
key_close	kclo	@3	KEY_CLOSE, Close key
key_command	kcmd	@4	KEY_COMMAND, Cmd (command) key
key_copy	kcpy	@5	KEY_COPY, Copy key
key_create	kcrt	@6	KEY_CREATE, Create key
key_ctab	kctab	kt	KEY_CTAB, Sent by clear-tab key
key_dc	kdch1	kD	KEY_DC, Sent by delete character key
key_dl	kdl1	kL	KEY_DL, Sent by delete line key
key_down	kcud1	kd	KEY_DOWN, Sent by terminal down arrow key
key_eic	krmir	kM	KEY_EIC, Sent by **rmir** or **smir** in insert mode
key_end	kend	@7	KEY_END, End key
key_enter	kent	@8	KEY_ENTER, Enter/send
key_eol	kel	kE	KEY_EOL, Sent by clear-to-end-of-line key
key_eos	ked	kS	KEY_EOS, Sent by clear-to-end-of-screen key
key_exit	kext	@9	KEY_EXIT, Sent by exit key
key_f0	kf0	k0	KEY_F(0), Sent by function key f0
key_f1	kf1	k1	KEY_F(1), Sent by function key f1
key_f2	kf2	k2	KEY_F(2), Sent by function key f2
key_f3	kf3	k3	KEY_F(3), Sent by function key f3
key_f4	kf4	k4	KEY_F(4), Sent by function key f4
key_f5	kf5	k5	KEY_F(5), Sent by function key f5
key_f6	kf6	k6	KEY_F(6), Sent by function key f6
key_f7	kf7	k7	KEY_F(7), Sent by function key f7
key_f8	kf8	k8	KEY_F(8), Sent by function key f8

Variable	Cap-name	Term-cap	Description Code
key_f9	kf9	k9	KEY_F(9), Sent by function key f9
key_f10	kf10	ka	KEY_F(10), Sent by function key f10
key_f11	kf11	F1	KEY_F(11), Sent by function key f11.
key_f12	kf12	F2	KEY_F(12), Sent by function key f12.
key_f13	kf13	F3	KEY_F(13), Sent by function key f13.
key_f14	kf14	F4	KEY_F(14), Sent by function key f14.
key_f15	kf15	F5	KEY_F(15), Sent by function key f15.
key_f16	kf16	F6	KEY_F(16), Sent by function key f16.
key_f17	kf17	F7	KEY_F(17), Sent by function key f17.
key_f18	kf18	F8	KEY_F(18), Sent by function key f18.
key_f19	kf19	F9	KEY_F(19), Sent by function key f19.
key_f20	kf20	FA	KEY_F(20), Sent by function key f20.
key_f21	kf21	FB	KEY_F(21), Sent by function key f21.
key_f22	kf22	FC	KEY_F(22), Sent by function key f22.
key_f23	kf23	FD	KEY_F(23), Sent by function key f23.
key_f24	kf24	FE	KEY_F(24), Sent by function key f24.
key_f25	kf25	FF	KEY_F(25), Sent by function key f25.
key_f26	kf26	FG	KEY_F(26), Sent by function key f26.
key_f27	kf27	FH	KEY_F(27), Sent by function key f27.
key_f28	kf28	FI	KEY_F(28), Sent by function key f28.
key_f29	kf29	FJ	KEY_F(29), Sent by function key f29.
key_f30	kf30	FK	KEY_F(30), Sent by function key f30.
key_f31	kf31	FL	KEY_F(31), Sent by function key f31.
key_f32	kf32	FM	KEY_F(32), Sent by function key f32.
key_f33	kf33	FN	KEY_F(33), Sent by function key f33.
key_f34	kf34	FO	KEY_F(34), Sent by function key f34.
key_f35	kf35	FP	KEY_F(35), Sent by function key f35.
key_f36	kf36	FQ	KEY_F(36), Sent by function key f36.
key_f37	kf37	FR	KEY_F(37), Sent by function key f37.
key_f38	kf38	FS	KEY_F(38), Sent by function key f38.
key_f39	kf39	FT	KEY_F(39), Sent by function key f39.
key_f40	kf40	FU	KEY_F(40), Sent by function key f40.
key_f41	kf41	FV	KEY_F(41), Sent by function key f41.
key_f42	kf42	FW	KEY_F(42), Sent by function key f42.
key_f43	kf43	FX	KEY_F(43), Sent by function key f43.
key_f44	kf44	FY	KEY_F(44), Sent by function key f44.
key_f45	kf45	FZ	KEY_F(45), Sent by function key f45.
key_f46	kf46	Fa	KEY_F(46), Sent by function key f46.
key_f47	kf47	Fb	KEY_F(47), Sent by function key f47.
key_f48	kf48	Fc	KEY_F(48), Sent by function key f48.
key_f49	kf49	Fd	KEY_F(49), Sent by function key f49.
key_f50	kf50	Fe	KEY_F(50), Sent by function key f50.

Variable	Cap-name	Term-cap	Description Code
key_f51	kf51	Ff	KEY_F(51), Sent by function key f51.
key_f52	kf52	Fg	KEY_F(52), Sent by function key f52.
key_f53	kf53	Fh	KEY_F(53), Sent by function key f53.
key_f54	kf54	Fi	KEY_F(54), Sent by function key f54.
key_f55	kf55	Fj	KEY_F(55), Sent by function key f55.
key_f56	kf56	Fk	KEY_F(56), Sent by function key f56.
key_f57	kf57	Fl	KEY_F(57), Sent by function key f57.
key_f58	kf58	Fm	KEY_F(58), Sent by function key f58.
key_f59	kf59	Fn	KEY_F(59), Sent by function key f59.
key_f60	kf60	Fo	KEY_F(60), Sent by function key f60.
key_f61	kf61	Fp	KEY_F(61), Sent by function key f61.
key_f62	kf62	Fq	KEY_F(62), Sent by function key f62.
key_f63	kf63	Fr	KEY_F(63), Sent by function key f63.
key_find	kfnd	@0	KEY_FIND, Sent by find key
key_help	khlp	%1	KEY_HELP, Sent by help key
key_home	khome	kh	KEY_HOME, Sent by home key
key_ic	kich1	kI	KEY_IC, Sent by ins char/enter ins mode key
key_il	kil1	kA	KEY_IL, Sent by insert line
key_left	kcub1	kl	KEY_LEFT, Sent by terminal left arrow key
key_ll	kll	kH	KEY_LL, Sent by home-down key
key_mark	kmrk	%2	KEY_MARK, Sent by mark key
key_message	kmsg	%3	KEY_MESSAGE, Sent by message key
key_move	kmov	%4	KEY_MOVE, Sent by move key
key_next	knxt	%5	KEY_NEXT, Sent by next object key
key_npage	knp	kN	KEY_NPAGE, Sent by next-page key
key_open	kopn	%6	KEY_OPEN, Sent by open key
key_options	kopt	%7	KEY_OPTIONS, Sent by options key
key_ppage	kpp	kP	KEY_PPAGE, Sent by previous-page key
key_previous	kprv	%8	KEY_PREVIOUS, Sent by previous object key
key_print	kprt	%9	KEY_PRINT, Sent by print or copy
key_redo	krdo	%0	KEY_REDO, Sent by redo key
key_reference	kref	&1	KEY_REFERENCE, Sent by ref(erence) key
key_refresh	krfr	&2	KEY_REFRESH, Sent by refresh key
key_replace	krpl	&3	KEY_REPLACE, Sent by replace key
key_restart	krst	&4	KEY_RESTART, Sent by restart key
key_resume	kres	&5	KEY_RESUME, Sent by resume key
key_right	kcuf1	kr	KEY_RIGHT, Sent by terminal right arrow key
key_save	ksav	&6	KEY_SAVE, Sent by save key
key_sbeg	kBEG	&9	KEY_SBEG, Sent by shifted beginning key
key_scancel	kCAN	&0	KEY_SCANCEL, Sent by shifted cancel key
key_scommand	kCMD	*1	KEY_SCOMMAND, Sent by shifted command key
key_scopy	kCPY	*2	KEY_SCOPY, Sent by shifted copy key

Variable	Cap-name	Term-cap	Description Code
key_screate	kCRT	*3	KEY_SCREATE, Sent by shifted create key
key_sdc	kDC	*4	KEY_SDC, Sent by shifted delete char key
key_sdl	kDL	*5	KEY_SDL, Sent by shifted delete line key
key_select	kslt	*6	KEY_SELECT, Sent by select key
key_send	kEND	*7	KEY_SEND, Sent by shifted end key
key_seol	kEOL	*8	KEY_SEOL, Sent by shifted clear line key
key_sexit	kEXT	*9	KEY_SEXIT, Sent by shifted exit key
key_sf	kind	kF	KEY_SF, Sent by scroll-forward/down key
key_sfind	kFND	*0	KEY_SFIND, Sent by shifted find key
key_shelp	kHLP	#1	KEY_SHELP, Sent by shifted help key
key_shome	kHOM	#2	KEY_SHOME, Sent by shifted home key
key_sic	kIC	#3	KEY_SIC, Sent by shifted input key
key_sleft	kLFT	#4	KEY_SLEFT, Sent by shifted left arrow key
key_smessage	kMSG	%a	KEY_SMESSAGE, Sent by shifted message key
key_smove	kMOV	%b	KEY_SMOVE, Sent by shifted move key
key_snext	kNXT	%c	KEY_SNEXT, Sent by shifted next key
key_soptions	kOPT	%d	KEY_SOPTIONS, Sent by shifted options key
key_sprevious	kPRV	%e	KEY_SPREVIOUS, Sent by shifted prev key
key_sprint	kPRT	%f	KEY_SPRINT, Sent by shifted print key
key_sr	kri	kR	KEY_SR, Sent by scroll-backward/up key
key_sredo	kRDO	%g	KEY_SREDO, Sent by shifted redo key
key_sreplace	kRPL	%h	KEY_SREPLACE, Sent by shifted replace key
key_sright	kRIT	%i	KEY_SRIGHT, Sent by shifted right arrow
key_srsume	kRES	%j	KEY_SRSUME, Sent by shifted resume key
key_ssave	kSAV	!1	KEY_SSAVE, Sent by shifted save key
key_ssuspend	kSPD	!2	KEY_SSUSPEND, Sent by shifted suspend key
key_stab	khts	kT	KEY_STAB, Sent by set-tab key
key_sundo	kUND	!3	KEY_SUNDO, Sent by shifted undo key
key_suspend	kspd	&7	KEY_SUSPEND, Sent by suspend key
key_undo	kund	&8	KEY_UNDO, Sent by undo key
key_up	kcuu1	ku	KEY_UP, Sent by terminal up arrow key
keypad_local	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit	smkx	ks	Put terminal in "keypad transmit" mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8

Variable	Cap-name	Term-cap	Description Code
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
meta_off	rmm	mo	(Reserved)
meta_on	smm	mm	(Reserved)
newline	nel	nw	Newline (like **cr** followed by **lf**)
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars (G*)
parm_delete_line	dl	DL	Delete #1 lines (G*)
parm_down_cursor	cud	DO	Move cursor down #1 lines (G*)
parm_ich	ich	IC	Insert #1 blank chars (G*)
parm_index	indn	SF	Scroll forward #1 lines (G)
parm_insert_line	il	AL	Add #1 new blank lines (G*)
parm_left_cursor	cub	LE	Move cursor left #1 spaces (G)
parm_right_cursor	cuf	RI	Move cursor right #1 spaces (G*)
parm_rindex	rin	SR	Scroll backward #1 lines (G)
parm_up_cursor	cuu	UP	Move cursor up #1 lines (G*)
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
plab_norm	pln	pn	Prog label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes.
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer
repeat_char	rep	rp	Repeat char #1 #2 times (G*)
req_for_input	rfi	RF	Send next input char (for pseudo-terminals)
reset_1string	rs1	r1	Reset terminal completely to sane modes
reset_2string	rs2	r2	Reset terminal completely to sane modes
reset_3string	rs3	r3	Reset terminal completely to sane modes
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last **sc**
row_address	vpa	cv	Vertical position absolute (G)
save_cursor	sc	sc	Save cursor position
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
set_attributes	sgr	sa	Define the video attributes #1-#9 (G)
set_left_margin	smgl	ML	Set soft left margin
set_right_margin	smgr	MR	Set soft right margin
set_tab	hts	st	Set a tab in all rows, current column
set_window	wind	wi	Current window: lines #1-#2 cols #3-#4

Variable	Cap-name	Term-cap	Description Code
tab	ht	ta	Tab to next 8 space hardware tab stop
to_status_line	tsl	ts	Go to status line, column #1
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)
xoff_character	xoffc	XF	X-off character
xon_character	xonc	XN	X-on character

A Sample Entry

The following entry, which describes the *Concept*-100 terminal, is among the more complex entries in the *terminfo* file.

```
concept100 | c100| concept | c104 | c100-4p | concept 100,
    am, db, eo, in, mir, ul, xenl,
    cols#80, lines#24, pb#9600, vt#8,
    bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>,
    cnorm=\Ew, cr=^M$<9>, cub1=^H, cud1=^J,
    cuf1=\E=, cup=\Ea%p1%' '%+%c%p2%' '%+%c,
    cuu1=\E;, cvvis=\EW, dch1=\E^A$<16*>, dim=\EE,
    dl1=\E^B$<3*>, ed=\E^C$<16*>, el=\E^U$<16>,
    flash=\Ek$<20>\EK, ht=\t$<8>, il1=\E^R$<3*>,
    ind=^J, .ind=^J$<9>, ip=$<16*>,
    is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\0\Eo&\0\Eo\47\E,
    kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
    kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
    prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
    rev=\ED, rmcup=\Ev\s\s\s\s$<6>\Ep\r\n,
    rmir=\E\0, rmkx=\Ex, rmso=\Ed\Ee, rmul=\Eg,
    rmul=\Eg, sgr0=\EN\0, smcup=\EU\Ev\s\s8p\Ep\r,
    smir=\E^P, smkx=\EX, smso=\EE\ED, smul=\EG,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Lines beginning with "#" are taken as comment lines. Capabilities in *terminfo* are of three types: boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular features, and string capabilities which give a sequence that can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the Concept has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value '80' for the Concept.

Finally, string-valued capabilities, such as **el** (clear to end of line sequence) are given by the two- to five-character capname, an '=', and then a string ending at the next following ','. A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el**=\EK$<3>, and padding characters are supplied by **tputs**( ) [see CURSES(TI_LIB)] to provide this delay. The delay can be either a number, e.g., '20', or a number followed by an '*', i.e., '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of *lines* affected. This is always one unless the terminal has **in** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.) If the terminal has **xon** defined, the padding information is advisory and will only be used for cost estimates or when the terminal is in raw mode.

A number of escape sequences are provided in the string-valued capabilities for easy encoding of characters there. Both **E** and **e** map to an ESCAPE character, ^**x** maps to a control-x for any appropriate x, and the sequences **n**, **l**, **r**, **t**, **b**, **f**, and **s** give a newline, linefeed, return, tab, backspace, formfeed, and space. Other escapes include \^ for caret(^), \\ for backslash (\), \, for comma (,); \: for colon (:), and **0** for null. Finally, characters may be given as three octal digits after a backslash (e.g., \123).

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above. Note that capabilities are defined in a left-to-right order and therefore, a prior definition will override a later definition.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os.** (**os** applies to storage scope terminals, such as Tektronix 4010 series, as well as hardcopy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr.** (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc.) give this as **bel.** If the terminal uses the xon-xoff flow-control protocol, like most terminals, specify **xon.**

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1.** Similarly, codes to move to the right, up, and down should be given as **cuf1, cuu1,** and **cud1.** These local cursor motions should not alter the text they pass over; for example, you would not normally use '**cuf1**=\s' because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am.** If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus the AT&T model 33 is described as

```
33 | tty33 | tty | model 33 teletype,
    bel=^G, cols#72, cr=^M, cud1=^J, hc,
    ind=^J, os,
```

while the Lear Siegler ADM-3 is described as

```
adm3 | lsi adm3,
    am, bel=^G, clear=^Z, cols#80, cr=^M,
    cub1=^H, cud1=^J, ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup.**

The parameter mechanism uses a stack and special % codes to manipulate it in the manner of a Reverse Polish Notation calculator. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary. Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use "%gx%{5}%-".

The % encodings have the following meanings:

%%	outputs '%'
%[[:]*flags*][*width*[.*precision*]][**doxXs**]	as in **printf**(BA_LIB), flags are **[−+#]** and space
%d	print pop() as a decimal number

<table>
<tr><td>%3d</td><td>print pop() in a field at least 3 spaces wide</td></tr>
<tr><td>%03d</td><td>use leading zeros to fill</td></tr>
<tr><td>%s</td><td>print pop() as a character string</td></tr>
<tr><td>%c</td><td>print pop() gives %c</td></tr>
<tr><td>%p[1-9]</td><td>push i^{th} parm</td></tr>
<tr><td>%P[a-z]</td><td>set variable [a-z] to pop()</td></tr>
<tr><td>%g[a-z]</td><td>get variable [a-z] and push it</td></tr>
<tr><td>%'c'</td><td>push char constant *c*</td></tr>
<tr><td>%{*nn*}</td><td>push decimal constant *nn*</td></tr>
<tr><td>%l</td><td>push strlen(pop())</td></tr>
<tr><td>%+ %- %* %/ %m</td><td>arithmetic (%m is mod): push(pop() op pop())</td></tr>
<tr><td>%& %| %^</td><td>bit operations: push(pop() op pop())</td></tr>
<tr><td>%= %> %<</td><td>logical operations: push(pop() op pop())</td></tr>
<tr><td>%A %O</td><td>logical operations: and, or</td></tr>
<tr><td>%! %~</td><td>unary operations: push(op pop())</td></tr>
<tr><td>%i</td><td>add 1 to first parm, if one parm present, or first two parms, if more than one parm present (for ANSI terminals)</td></tr>
</table>

%? expr %t thenpart %e elsepart %;	if-then-else, %e elsepart is optional; else-if's are possible: %? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e b_5%; c_i are conditions, b_i are bodies.

If the "-" flag is used with "%[doxXs]", then a colon (:) must be placed between the "%" and the "-" to differentiate the flag from the binary "%-" operator, .e.g "%:-16.16s".

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is **cup**=\E&a%p2%02dc%p1%02dY$<6>.

The Micro-Term ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, **cup**=^T%p1%c%p2%c. Terminals which use %c need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **n**, **^D**, and **r**, as the system may change or discard them. (The library routines dealing with *terminfo* set tty modes so that tabs are never expanded, so **t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus **cup**=\E=%p1%'\s'%+%c%p2%'\s'%+%c. After sending '\E=', this pushes the first parameter, pushes the **ASCII** value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory.

(Thus, the **EH** sequence on Hewlett-Packard terminals cannot be used for **home** without losing some of the other features of the terminal.)

If the terminal has row or column absolute cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two parameter sequence (as with the HP2645) and can be used in preference to **cup.** If there are parameterized local motions (e.g., move *n* spaces to the right), these can be given as **cud, cub, cuf,** and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup,** such as the Tektronix 4025.

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the beginning of the line to the current position inclusive, leaving the cursor where it is, this should be given as **el1**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. The **ed** capability is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/Delete Line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1**, which take a single parameter and insert or delete that many lines, can be given as **il** and **dl.**

If the terminal has a settable destructive scrolling region (like the VT100), the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is undefined after using this command. It is possible to get the effect of insert or delete line using this command — the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

To determine whether a terminal has destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index **ri** followed by a delete line **dl1** or index **ind**. If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the **dl1** or **ind**, then the terminal has non-destructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify **csr** if the terminal has non-destructive scrolling regions, unless **ind**, **ri**, **indn**, **rin**, **dl**, and **dl1** all simulate destructive scrolling.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind.** The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *terminfo*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc   def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. While these are two logically separate attributes (one line vs. multiline insert mode and special treatment of untyped spaces), we have seen no terminals whose insert mode cannot be described with the single attribute.

The *terminfo* database can describe both terminals which have an insert mode and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode.

Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n,* will repeat the effects of **ich1** *n* times.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in **rmp**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n,* to delete *n* characters and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode,* representing a good, high contrast, easy-on-the-eyes format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half-bright is good, or reverse video alone; however, different users have different preferences on different terminals.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TeleVideo 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul**, respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Micro-Term MIME, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking), **bold** (bold or extra bright), **dim** (dim or half-bright), **invis** (blanking or invisible text), **prot** (protected), **rev** (reverse video), **sgr0** (turn off *all* attribute modes), **smacs** (enter alternate character set mode), and **rmacs** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes. If a command is necessary before alternate character set mode is entered, give the sequence in **enacs** (enable alternate-character-set mode).

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine parameters. Each parameter is either 0 or non-zero, as the corresponding attribute is on or off. The nine parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, and alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **flash**; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad 200 ms, then return the screen to normal video.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis.** The boolean **chts** should also be given. If there is a way to make the cursor completely invisible, give that as **civis.** The capability **cnorm** should be given, which undoes the effects of both of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory-relative cursor addressing and not screen-relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by *terminfo*. If the **smcup** sequence will not restore the screen after an **rmcup** sequence is output (to the state prior to outputting **rmcup**), specify **nrrmc**.

If your terminal correctly generates underlined characters by using the underline character (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. For terminals where a character overstriking another displays both characters (typically a hardcopy terminal), give the capability **os**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise, the keypad is assumed to always transmit.

The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome**, respectively. If there are function keys such as f0, f1, ..., f63, the codes they send can be given as **kf0, kf1, ..., kf63**. If the first 11 keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3-by-3 array of keys including the four arrow keys, the other five keys can be given as **ka1, ka3, kb2, kc1,** and **kc3.** These keys are useful when the effects of a 3-by-3 directional pad are needed. Additional keys are defined above in the capabilities list.

Strings to program function keys can be given as **pfkey, pfloc,** and **pfx.** A string to program soft-screen labels can be given as **pln**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal-dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local; and **pfx** causes the string to be transmitted to the computer. The capabilities **nlab**, **lw**, and **lh** define how many soft labels there are and their width and height. If there are commands to turn the labels on and off, give them in **smln** and **rmln**; **smln** is normally output after one or more **pln** sequences to make sure that the change becomes visible.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt.** By convention, if the terminal modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by **tput init** [see TPUT(TI_CMD)] to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the *terminfo* description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row).

Other capabilities include **is1, is2,** and **is3,** initialization strings for the terminal, **iprog,** the pathname of a program to be run to initialize the terminal, and **if,** the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the *terminfo* description. They must be sent to the terminal each time the user logs in and be output in the following order: run the program **iprog**, output **is1**; **is2;** set the margins using **mgc**, **smgl** and **smgr**; set tabs using **tbc** and **hts;** print the file **if;** and finally output **is3.** This can be done using the **init** argument of the TPUT(TI_CMD) command.

Most initialization is done with **is2.** Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3.** A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1, rs2, rf,** and **rs3,** analogous to **is*** and **if.** (The method using files, **if** and **rf**, is not recommended; the recommended method is to use the initialization and reset strings.) These strings should be output when the terminal gets into an unreasonable state by using the **reset** argument to the TPUT(TI_CMD) command. Commands are normally placed in **rs*** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2,** but on some terminals it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80-column mode. Therefore, the command is usually placed in **rs1**, not **is2,** for those terminals.

If a more complex sequence is needed to set the tabs than can be described by using **tbc** and **hts**, the sequence can be placed in **is2** or **if.**

If there are commands to set and clear margins, they can be given as **mgc** (clear all margins), **smgl** (set left margin), and **smgr** (set right margin).

Delays

Certain capabilities control padding in the tty driver. These are primarily needed by hardcopy terminals, and are used by **tput init** to set tty modes appropriately. Delays embedded in the capabilities **cr, ind, cub1, ff,** and **tab** can be used to set the appropriate delay bits in the tty driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb.**

Status Line

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl.** (**fsl** must leave the cursor position in the same place it was before **tsl.** If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as **tab** and **el**, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl.** If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc.** The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols.** If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl.**

Line Graphics

If the terminal has a line drawing alternate character set, the mapping of glyph to character would be given in **acsc**. The definition of this string is based on the alternate character set used in the DEC VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

glyph name	vt100+ character
arrow pointing right	+
arrow pointing left	,
arrow pointing down	.
solid square block	0
lantern symbol	I
arrow pointing up	-
diamond	`
checker board (stipple)	a
degree symbol	f
plus/minus	g
board of squares	h
lower right corner	j
upper right corner	k
upper left corner	l
lower left corner	m
plus	n
scan line 1	o
horizontal line	q
scan line 9	s
left tee (├)	t
right tee (┤)	u
bottom tee (┴)	v
top tee (┬)	w
vertical line	x
bullet	~

The best way to describe a new terminal's line graphics set is to add a third column to the above table with the characters for the new terminal that produce the appropriate glyph when the terminal is in the alternate character set mode. For example,

glyph name	vt100+ char	new tty char
upper left corner	l	R
lower left corner	m	F
upper right corner	k	T
lower right corner	j	G
horizontal line	q	,
vertical line	x	.

Now, write down the characters left to right, as in "**acsc**=lRmFkTjGq\,x.".

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used. If the terminal does not have a pad character, specify **npc**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep.** The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, **tparm(repeat_char, 'x', 10)** is the same as 'xxxxxxxxxx'.

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch.** A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it.

Terminal descriptions that do not represent a specific kind of known terminal, such as **switch, dialup, patch,** and **network,** should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. A line-turn-around sequence to be transmitted before doing reads should be specified in **rfi.**

If the terminal uses xon/xoff handshaking for flow control, give **xon.** Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted. Sequences to turn on and off xon/xoff handshaking may be given in **smxon** and **rmxon**. If the characters used for handshaking are not ^S and ^Q, they may be specified with **xonc** and **xoffc**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm.** A value of **lm**#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. If the text is not displayed on the terminal screen when the printer is on, specify **mc5i** (silent printer). All text, including **mc4,** is transparently passed to the printer while an **mc5p** is in effect.

Special Cases

The working model used by *terminfo* fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by *terminfo*. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware.

Terminals which can not display **tilde** characters, such as certain Hazeltine terminals, should indicate **hz.**

Terminals which ignore a linefeed immediately after an **am** wrap, such as the Concept 100, should indicate **xenl.** Those terminals whose cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate **xenl.**

If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given.

Those Teleray terminals whose tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie"; therefore, to erase standout mode it is instead necessary to use delete and insert line.

The Beehive Superbee terminals, which do not transmit the escape or control C characters, should specify **xsb,** indicating that the f1 key is to be used for escape and f2 for control C.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use.** A capability can be cancelled by placing **xx@** to the left of the capability definition, where **xx** is the capability. For example, the entry

```
att4424-2   |AT&T   4424   in   display   function
group ii,
        rev@, sgr@, smul@, use=att4424,
```

defines an AT&T 4424 terminal that does not have the **rev**, **sgr**, and **smul** capabilities, and hence cannot do highlighting. This is useful for different modes for a terminal, or for different user preferences. More than one **use** capability may be given.

FILES

/usr/lib/terminfo/?/* Compiled terminal description database

SEE ALSO

CURSES(TI_LIB), PRINTF(BA_LIB), TIC(TI_CMD).

USAGE

Administrator and Application Program.

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with VI(AU_CMD) to check that they are correct. To easily test a new terminal description the environment variable **TERMINFO** can be set to the pathname of a directory containing the compiled description, and programs will look there rather than in **/usr/lib/terminfo**. To get the padding for insert line right, a severe test is to comment out **xon**, edit a copy of a large file at 9600 baud with VI(AU_CMD), delete 16 or so lines from the middle of the screen, then hit the 'u' key several times quickly. If the terminal messes up, more padding is usually needed. A similar test can be used for insert character.

LEVEL

Level 1.

Chapter 10
Library Routines

NAME

curses - CRT screen handling and optimization package

SYNOPSIS

#include <curses.h>

DESCRIPTION

The *curses* library routines give the user a terminal-independent method of updating screens with reasonable optimization. A program using these routines must be compiled with the **-lcurses** option of **cc**.

In order to initialize the routines, the routine **initscr**( ) or **newterm**( ) must be called before any of the other routines that deal with windows and screens are used. The routine **endwin**( ) must be called before exiting. To get character-at-a-time input without echoing (most interactive, screen oriented-programs want this), the following sequence should be used:

```
initscr(), cbreak(),  noecho();
```

Most programs would additionally use the sequence:

```
nonl(), intrflush(stdscr, FALSE);
    keypad(stdscr, TRUE).
```

Before a *curses* program is run, a terminal's tabs stops should be set and its initialization strings, if defined, must be output. This can be done by executing the **tput init** command after the shell environment variable **TERM** has been exported. See TERMINFO(TI_ENV) for further details.

The *curses* library permits manipulation of data structures called *windows* which can be thought of as two-dimensional arrays of characters representing all or part of a CRT screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others may be created with **newwin**( ). Windows are referred to by variables declared as "**WINDOW** *". These data structures are manipulated with routines described below, among which the most basic are **move**( ) and **addch**( ). (More general versions of these routines are included with names beginning with **w**, allowing one to specify a window. The routines not beginning with **w** affect **stdscr**.) Then **refresh**( ) is called, telling the routines to make the user's CRT screen look like **stdscr.** The characters in a window are actually of type **chtype**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be completely displayed. See the description of **newpad**( ) under "Window and Pad Manipulation" for more information.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to show up in such modes as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters may be specifed to be output. On input, *curses* is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in <**curses.h**>, such as **A_REVERSE**, **ACS_HLINE**, and **KEY_LEFT**.

The environment variables **LINES** and **COLUMNS** may also be set to override *terminfo*'s idea of how large a screen is. These may be used in an AT&T 5620 layer, for example, where the size of a screen is changeable.

If the environment variable **TERMINFO** is defined, any program using *curses* will check for a local terminal definition before checking in the standard place. For example, if TERM is set to **"att4424"**, then the compiled terminal definition is found in **/usr/lib/terminfo/a/att4424**. (The "a" is copied from the first letter of "att4424" to avoid creation of huge directories.) However, if **TERMINFO** is set to **$HOME/myterms**, *curses* will first check **$HOME/myterms/a/att4424**, and if that fails, will then check **/usr/lib/terminfo/a/att4424**. This is useful for developing experimental definitions or when write permission in **/usr/lib/terminfo** is not available.

The integer variables **LINES** and **COLS** are defined in <**curses.h**> and will be filled in by **initscr**( ) with the size of the screen. The constants **TRUE** and **FALSE** have the values **1** and **0**, respectively.

The *curses* routines also define the **WINDOW** * variable **curscr** which is used for certain low-level operations like clearing and redrawing a garbaged screen. The **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen is immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" function. More information on using **curscr** is provided where its use is appropriate.

Routines

Many of the following routines have two or more versions. The routines prefixed with **w** require a **window** argument. The routines prefixed with **p** require a **pad** argument. Those without a prefix generally use **stdscr.**

The routines prefixed with **mv** require an **x** and **y** coordinate to move to before performing the appropriate action. The **mv** routines imply a call to **move** before the call to the other routine. The coordinate **y** always refers to the row (of the window), and **x** always refers to the column. The upper left corner is always (0,0), not (1,1).

The routines prefixed with **mvw** take both a **window** argument and **x** and **y** coordinates. The window argument is always specified before the coordinates.

In each case, **win** is the window affected and **pad** is the pad affected; **win** and **pad** are always of type **WINDOW**. Option setting routines require a boolean flag **bf** with the value **TRUE** or **FALSE**; **bf** is always of type **bool**. The variables **ch** and **attrs** below are always of type **chtype.** The types **WINDOW**, **bool**, and **chtype** are defined in <**curses.h**>. All other arguments are integers.

See the **RETURN VALUE** paragraph near the end of the TI_LIB section for information on the values returned by the routines described below.

Overall Screen Manipulation

WINDOW ***initscr**( )

The first routine called should almost always be **initscr**( ). (The exceptions are **slk_init**( ), **filter**( ), and **ripoffline**().) This will determine the terminal type and initialize all *curses* data structures. **initscr**( ) also arranges that the first call to **refresh**( ) will clear the screen. If errors occur, **initscr**( ) will write an appropriate error message to standard error and exit; otherwise, a pointer to **stdscr**( ) is returned. If the program wants an indication of error conditions, **newterm**() should be used instead of **initscr**( ) — **initscr**( ) should only be called once per application.

endwin( )

A program should always call **endwin**( ) before exiting or escaping from *curses* mode temporarily. This routine will restore tty modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call **refresh**( ) or **doupdate**( ).

SCREEN ***newterm(type, outfd, infd)**
char *type;
FILE ***outfd, *infd;**

A program which outputs to more than one terminal should use **newterm**( ) for each terminal instead of **initscr**( ). A program which wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use this routine. The routine

newterm( ) should be called once for each terminal. It returns a variable of type **SCREEN** * which should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of **TERM,** a file pointer for output to the terminal, and another file pointer for input from the terminal. The program must also call **endwin**( ) for each terminal being used before exiting from curses. If **newterm**( ) is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin**( ) is called.

SCREEN *set_term(new)
SCREEN *new;

This routine is used to switch between different terminals. The screen reference **new** becomes the new current terminal. The previous terminal is returned by the routine. This is the only routine which manipulates **SCREEN** pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

refresh( )

wrefresh(win)
WINDOW *win;

These routines (or **prefresh**( ), **pnoutrefresh**( ), **wnoutrefresh**( ), or **doupdate**( )) must be called to get any output on the terminal, as other routines merely manipulate data structures. The routine **wrefresh**( ) copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. The **refresh**( ) routine is the same, using **stdscr** as a default screen. Unless **leaveok**( ) has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

NOTE: **refresh** is a macro.

wnoutrefresh(win)
WINDOW *win;

doupdate( )

These two routines allow multiple updates with more efficiency than **wrefresh**( ) alone. In addition to all of the window structures, *curses* keeps two data structures representing the terminal screen: a **physical** screen, describing what is actually on the screen, and a **virtual** screen, describing what the programmer **wants** to have on the screen.

The routine **wrefresh**( ) works by first calling **wnoutrefresh**( ), which copies the named window to the virtual screen, and then calling

doupdate( ), which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh**( ) will result in alternating calls to **wnoutrefresh**( ) and **doupdate**( ), causing several bursts of output to the screen. By first calling **wnoutrefresh**( ) for each window, it is then possible to call **doupdate**( ) once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less CPU time used.

WINDOW ***newwin(nlines, ncols, begin_y, begin_x)**
int nlines, ncols, begin_y, begin_x;

Create and return a pointer to a new window with the given number of lines, **nlines**, and columns, **ncols**. The upper left corner of the window is at line **begin_y**, column **begin_x.** If either **nlines** or **ncols** is zero, they will be defaulted to **LINES — begin_y** and **COLS — begin_x.** A new full-screen window is created by calling **newwin**(0,0,0,0).

mvwin(win, y, x)
WINDOW *win;
int y, x;

Move the window so that the upper left corner will be at position (**x**, **y**). If the move would cause the window to be off the screen, it is an error and the window is not moved. Moving subwindows is allowed, but should be avoided.

WINDOW ***subwin(orig, nlines, ncols, begin_y, begin_x)**
WINDOW ***orig;**
int nlines, ncols, begin_y, begin_x;

Create and return a pointer to a new window with the given number of lines, **nlines**, and columns, **ncols**. The window is at position (**begin_y, begin_x**) on the screen. (This position is relative to the screen, and not to the window **orig**.) The window is made in the middle of the window **orig,** so that changes made to one window will affect both windows. The subwindow shares memory with the window **orig**. When using this routine, it will be necessary to call **touchwin**( ) or **touchline**( ) on **orig** before calling **wrefresh**( ) on the subwindow.

delwin(win)
WINDOW *win;

Deletes the named window, freeing up all memory associated with it. Subwindows must be deleted before the main window.

WINDOW *newpad(nlines, ncols)
int nlines, ncols;

Create and return a pointer to a new **pad** data structure with the given number of lines, **nlines**, and columns, **ncols**. A **pad** is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g., from scrolling or echoing of input) do not occur. It is not legal to call **wrefresh**( ) with a **pad** as an argument; the routines **prefresh**( ) or **pnoutrefresh**( ) should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW *subpad(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;

Create and return a pointer to a subwindow within a pad with the given number of lines, **nlines**, and columns, **ncols**. Unlike **subwin**( ), which uses screen coordinates, the window is at position (**begin_x, begin_y**) on the pad. The window is made in the middle of the window **orig**, so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call **touchwin**( ) or **touchline**( ) on **orig** before calling **prefresh**( ).

prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
WINDOW *pad;
int pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol;

pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
WINDOW *pad;
int pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol;

These routines are analogous to **wrefresh**( ) and **wnoutrefresh**( ) except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. **pminrow** and **pmincol** specify the upper left corner, in the pad, of the rectangle to be displayed. **sminrow, smincol, smaxrow,** and **smaxcol** specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of **pminrow, pmincol, sminrow**, or **smincol** are treated as if they were zero.

Output

These routines are used to "draw" text on windows.

addch(ch)
chtype ch;

waddch(win, ch)
WINDOW *win;
chtype ch;

mvaddch(y, x, ch)
int y, x;
chtype ch;

mvwaddch(win, y, x, ch)
WINDOW *win;
int y, x;
chtype ch;

The character **ch** is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of **putchar**. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok**( ) is enabled, the scrolling region will be scrolled up one line.

If **ch** is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a **clrtoeol**( ) before moving. Tabs are considered to be at every eighth column. If **ch** is another control character, it will be drawn in the ^X notation. Calling **winch**( ) after adding a control character will not return the control character, but instead will return the representation of the control character.

Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **inch**( ) and **addch**( ).) See **standout**( ) below.

NOTE: **addch**, **mvaddch**, and **mvwaddch** are macros.

echochar(ch)
chtype ch;

wechochar(win, ch)
WINDOW *win;
chtype ch;

pechochar(pad, ch)
WINDOW *pad;
chtype ch;

These routines are functionally equivalent to a call to **addch(ch)** followed by a call to **refresh**( ), a call to **waddch(win, ch)** followed by a call to **wrefresh(win)**, or a call to **waddch(pad, ch)** followed by a call to **prefresh(pad).** The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of **pechochar**( ), the last location of the pad on the screen is reused for the arguments to **prefresh**( ).

NOTE: **echochar**( ) is a macro.

addstr(str)
char *str;

waddstr(win, str)
WINDOW *win;
char *str;

mvaddstr(y, x, str)
int y, x;
char *str;

mvwaddstr(win, y, x, str)
WINDOW *win;
int y, x;
char *str;

These routines write all the characters of the null terminated character string **str** on the given window. It is equivalent to calling **waddch**( ) once for each character in the string.

NOTE: **addstr**, **mvaddstr**, and **mvwaddstr** are macros.

attroff(attrs)
int attrs;

wattroff(win, attrs)
WINDOW *win;
int attrs;

attron(attrs)
int attrs;

wattron(win, attrs)
WINDOW *win;
int attrs;

attrset(attrs)
int attrs;

wattrset(win, attrs)
WINDOW *win;
int attrs;

standend()

wstandend(win)
WINDOW *win;

standout()

wstandout(win)
WINDOW *win;

These routines manipulate the *current attributes* of the named window. These attributes can be any combination of **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, **A_UNDERLINE**, and **A_ALTCHARSET**. These constants are defined in <**curses.h**> and can be combined with the C | (or) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch()**. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of characters put on the screen.

The routine **attrset(attrs)** sets the current attributes of the given window to **attrs**. **attroff(attrs)** turns off the named attributes without turning on or off any other attributes. **attron(attrs)** turns on the named attributes without affecting any others. **standout()** is the same as **attron(A_STANDOUT)**. **standend()** is the same as **attrset**(0), that is, it turns off all attributes.

NOTE: **attroff**, **attron**, **attrset**, **standend** and **standout** are macros.

beep( )

flash( )

These routines are used to signal the terminal user. **beep( )** will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash( )** will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep), but only some can flash the screen.

box(win, vert, hor)
WINDOW *win;
chtype vert, hor;

A box is drawn around the edge of the window. **vert** and **hor** are the characters the box is to be drawn with. If **vert** and **hor** are 0, then appropriate default characters, **ACS_VLINE** and **ACS_HLINE**, will be used.

erase( )

werase(win)
WINDOW *win;

These routines copy blanks to every position in the window.

NOTE: **erase** is a macro.

clear( )

wclear(win)
WINDOW *win;

These routines are like **erase( )** and **werase( )**, but they also call **clearok( )**, arranging that the screen will be cleared completely on the next call to **wrefresh( )** for that window and repainted from scratch.

NOTE: **clear** is a macro.

clrtobot( )

wclrtobot(win)
WINDOW *win;

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

NOTE: **clrtobot** is a macro.

clrtoeol()
wclrtoeol(win)
WINDOW *win;

The current line to the right of the cursor, inclusive, is erased.

NOTE: **clrtoeol** is a macro.

delay_output(ms)
int ms;

Insert **ms** millisecond pause in output. It is not recommended that this routine be used extensively since padding characters are used rather than a CPU pause.

delch()
wdelch(win)
WINDOW *win;
mvdelch(y, x)
int y, x;
mvwdelch(win, y, x)
WINDOW *win;
int y, x;

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to **y**, **x**, if specified). (This does not imply use of the hardware delete character feature.)

NOTE: **delch**, **mvdelch**, and **mvwdelch** are macros.

deleteln()
wdeleteln(win)
WINDOW *win;

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware delete line feature.)

NOTE: **deleteln** is a macro.

getyx(win, y, x)
WINDOW *win;
int y, x;

The cursor position of the window is placed in the two integer variables **y** and **x.** This is implemented as a macro, so no & is necessary before the variables.

NOTE: **getyx** is a macro.

getbegyx(win, y, x)
WINDOW *win;
int y, x;

getmaxyx(win, y, x)
WINDOW *win;
int y, x;

Like **getyx**( ), these routines store the current beginning coordinates and size of the specified window.

NOTE: **getbegyx** and **getmaxyx** are macros.

insch(ch)
chtype ch;

winsch(win, ch)
WINDOW *win;
chtype ch;

mvinsch(y, x, ch)
int y, x;
chtype ch;

mvwinsch(win, y, x, ch)
WINDOW *win;
int y, x;
chtype ch;

The character **ch** is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. The cursor position does not change (after moving to **y**, **x**, if specified). (This does not imply use of the hardware insert character feature.)

NOTE: **insch**, **mvinsch**, and **mvwinsch** are macros.

insertln()

winsertln(win)
WINDOW *win;

A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware insert line feature.)

NOTE: **insertln** is a macro.

move(y, x)

wmove(win, y, x)
WINDOW *win;
int y, x;

The cursor associated with the window is moved to line **y** and column **x**. This does not move the physical cursor of the terminal until **refresh()** is called. The position specified is relative to the upper left corner of the window, which is (0,0).

NOTE: **move** is a macro.

overlay(srcwin, dstwin)
WINDOW *srcwin, *dstwin;

overwrite(srcwin, dstwin)
WINDOW *srcwin, *dstwin;

These routines overlay **srcwin** on top of **dstwin. Scrwin** and **dstwin** are not required to be the same size; only text where the two windows overlap is copied. The difference is that **overlay()** is non-destructive (blanks are not copied) while **overwrite()** is destructive.

copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
WINDOW *srcwin, *dstwin;
int sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay;

This routine provides a finer grain of control over the **overlay()** and **overwrite()** routines. Like in the **prefresh()** routine, a rectangle is specified in the destination window, **(dminrow, dmincol)** and **(dmaxrow, dmaxcol)**, and the upper-left-corner coordinates of the source window, **(sminrow, smincol)**. If the argument **overlay** is true, then copying is non-destructive, as in **overlay()**.

printw(fmt [, arg] ...)
char *fmt;

wprintw(win, fmt [, arg] ...)
WINDOW *win;
char *fmt;

mvprintw(y, x, fmt [, arg] ...)
int y, x;
char *fmt;

mvwprintw(win, y, x, fmt [, arg] ...)
WINDOW *win;
int y, x;
char *fmt;

These routines are analogous to **printf** [see PRINTF(BA_LIB)]. The string which would be output by **printf** is instead output using **waddstr**( ) on the given window.

scroll(win)
WINDOW *win;

The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window's scrolling region is the entire screen, the physical screen will be scrolled at the same time.

touchwin(win)
WINDOW *win;

touchline(win, start, count)
WINDOW *win;
int start, count;

Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change. **touchline**( ) only pretends that **count** lines have been changed, beginning with line **start**.

Input

The following routines are used to obtain input from windows.

getch( )
wgetch(win)
WINDOW *win;
mvgetch(y, x)
int y, x;
mvwgetch(win, y, x)
WINDOW *win;
int y, x;

A character is read from the terminal associated with the window. In nodelay mode, if there is no input waiting, the value ERR is returned. In delay mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**( ), this will be after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program will hang until a character is typed or the specified timeout has been reached. Unless **noecho**( ) has been set, the character will also be echoed into the designated window. No **refresh**( ) will occur between the **move**( ) and the **getch**( ) done within the routines **mvgetch**( ) and **mvwgetch**( ).

When using **getch**( ), **wgetch**( ), **mvgetch**( ), or **mvwgetch**( ), do not set both NOCBREAK mode (**nocbreak**()) and ECHO mode (**echo**()) at the same time. Depending on the state of the tty driver when each character is typed, the program may produce undesirable results.

If **keypad**( ) is **TRUE**, and a function key is pressed, the token for that function key will be returned instead of the raw characters. Possible function keys are defined in <**curses.h**> with integers beginning with 0401, whose names begin with **KEY_**. If a character is received that could be the beginning of a function key (such as escape), *curses* will set a timer. If the remainder of the sequence does not come in within the designated time, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of the escape key for a single character function is discouraged.) Since tokens returned by these routines are outside of the ASCII range, they are not printable.

NOTE: getch, **mvgetch**, and **mvwgetch** are macros.

getstr(str)
char *str;

wgetstr(win, str)
WINDOW *win;
char *str;

mvgetstr(y, x, str)
int y, x;
char *str;

mvwgetstr(win, y, x, str)
WINDOW *win;
int y, x;
char *str;

A series of calls to **getch()** is made, until a newline and carriage return is received. The resulting value is placed in the area pointed at by the character pointer **str.** The user's erase and kill characters are interpreted.

NOTE: **getstr**, **mvgetstr**, and **mvwgetstr** are macros.

flushinp()

Throws away any typeahead that has been typed by the user and has not yet been read by the program.

ungetch(ch)
chtype ch;

Place **ch** back onto the input queue to be returned by the next call to **wgetch()**.

chtype inch()

chtype winch(win)
WINDOW *win;

chtype mvinch(y, x)
int y, x;

chtype mvwinch(win, y, x)
WINDOW *win;
int y, x;

The character, of type **chtype,** at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants **A_CHARTEXT** and **A_ATTRIBUTES**, defined in **<curses.h>**, can be used with the **&** (logical and) operator to extract the character or attributes alone.

NOTE: **inch**, **winch**, **mvinch**, and **mvwinch** are macros.

scanw(fmt [, arg] ...)
char *fmt;

wscanw(win, fmt [, arg] ...)
WINDOW *win;
char *fmt;

mvscanw(y, x, fmt [, arg] ...)
int y, x;
char *fmt;

mvwscanw(win, y, x, fmt [, arg] ...)
WINDOW *win;
int y, x;
char *fmt;

These routines correspond to **scanf** [see SCANF(BA_LIB)]. The **wgetstr**( ) is called on the window, and the resulting line is used as input for the scan. Fields which do not map to a variable in the **fmt** field are lost. Users may interrogate the return value from these routines to determine the number of fields which were mapped in the call.

Output Options Setting

These routines set options within *curses* that deal with output. All options are initially **FALSE**, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**( ).

clearok(win, bf)
WINDOW *win;
bool bf;

If enabled (**bf** is **TRUE**,) the next call to **wrefresh**( ) with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win, bf)
WINDOW *win;
bool bf;

If enabled (**bf** is **TRUE**), *curses* will consider using the hardware insert/delete line feature of terminals so equipped. If disabled (**bf** is **FALSE**), *curses* will very seldom use this feature. (The insert/delete character feature is always considered.) This option should be enabled only if the application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If insert/delete line cannot be used, *curses* will redraw the changed portions of all lines.

leaveok(win, bf)
WINDOW *win;
bool bf;

Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

setscrreg(top, bot)
int top, bot;

wsetscrreg(win, top, bot)
WINDOW *win;
int top, bot;

These routines allow the user to set a software scrolling region in a window. **top** and **bot** are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok( )** are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. Only the text of the window is scrolled. (Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the VT100. If **idlok( )** is enabled and the terminal has either a scrolling region or insert/delete line capability, they will probably be used by the output routines.)

scrollok(win, bf)
WINDOW *win;
bool bf;

This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled, (**bf** is **FALSE**), the cursor is left on the bottom line. If enabled, (**bf** is **TRUE**), **wrefresh( )** is called on the window, and then the physical terminal and window are scrolled up one line. [Note that in order to get the physical scrolling effect on the terminal, is is also necessary to call **idlok( )**.]

nl( )

nonl( )

These routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations using **nonl( )**, *curses* is able to make better use of the linefeed capability, resulting in faster cursor motion.

NOTE: nl is a macro.

Input Options Setting

cbreak( )

nocbreak( )

These two routines put the terminal into and out of CBREAK mode, respectively. In this mode, characters typed by the user are immediately available to the program and erase/kill character-processing is not performed. When out of this mode, the tty driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal may or may not be in CBREAK mode, as it is inherited; therefore, a program should call **cbreak( )** or **nocbreak( )** explicitly. Most interactive programs using *curses* will set this mode.

Note that **cbreak( )** overrides **raw( )**. See **getch( )** under "Input" for a discussion of how these routines interact with **echo( )** and **noecho( )**.

def_prog_mode( )

def_shell_mode( )

saveterm( )

Save the current terminal modes as the "program" (in *curses*) or "shell" (not in *curses*) state for use by the **reset_prog_mode( )** and **reset_shell_mode( )** routines. This is done automatically by **initscr( )**.

NOTE: The **saveterm( )** routine is being replaced by **def_prog_mode( )**, which provides the same functionality. **Saveterm( )** is included here for compatibility and is supported at level 2.

echo( )

noecho( )

These routines control whether characters typed by the user are echoed by **getch( )** as they are typed. Echoing by the tty driver is always disabled, but initally **getch( )** is in ECHO mode, so characters typed are echoed. Initially, characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling **noecho( )**. See **getch( )** under "Input" for a discussion of how these routines interact with **cbreak( )** and **nocbreak( )**.

halfdelay(tenths)
int tenths;

HALF-DELAY mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for **tenths** tenths of seconds, ERR will be returned if nothing has been typed. **tenths** must be a number between 1 and 255. Use **nocbreak( )** to leave HALF-DELAY mode.

intrflush(win, bf)
WINDOW *win;
bool bf;

If this option is enabled (**bf** is **TRUE**), when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing *curses* to have the wrong idea of what is on the screen. Disabling (**bf** is **FALSE**), the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

keypad(win, bf)
WINDOW *win;
bool bf;

This option enables the keypad of the user's terminal. If enabled (**bf** is **TRUE**), the user can press a function key (such as an arrow key) and **wgetch( )** will return a single value representing the function key, as in **KEY_LEFT**. (See **FUNCTION KEYS** below.) If disabled (**bf** is **FALSE**), *curses* will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will cause the terminal keypad to be turned on when **wgetch( )** is called. The default value for keypad is false.

nodelay(win, bf)
WINDOW *win;
bool bf;

This option causes **getch**( ) to be a non-blocking call. If no input is ready, **getch**( ) will return **ERR**. If disabled (**bf** is **FALSE**), **getch**( ) will hang until a key is pressed.

raw( )

noraw( )

The terminal is placed into or out of RAW mode. RAW mode is similar to CBREAK mode, in that characters typed are immediately passed through to the user program. The differences are that in RAW mode, the interrupt, quit, suspend and flow control characters are passed through uninterpreted, instead of generating a signal. The behavior of the BREAK key depends on other bits in the tty driver that are not set by *curses*.

reset_prog_mode( )

reset_shell_mode( )

fixterm( )

resetterm( )

Restore the terminal to "program" (in *curses*) or "shell" (out of *curses*) state. These are done automatically by **endwin**( ) and **doupdate**( ) after an **endwin**( ), so they would normally not be called before.

NOTE: The **fixterm**( ) routine is being replaced by **reset_prog_mode**( ) and the **resetterm**( ) routine is being replaced by **reset_shell_mode**( ). **fixterm**( ) and **resetterm**( ) are included here for compatibility and are supported at level 2.

resetty( )

savetty( )

These routines save and restore the state of the terminal modes. **savetty**( ) saves the current state in a buffer and **resetty**( ) restores the state to what it was at the last call to **savetty**( ).

typeahead(fd)
int fd;

Curses does “line-breakout optimization” by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update will be postponed until **refresh**( ) or **doupdate( )** is called again. This allows faster response to commands typed in advance. Normally, the input FILE pointer passed to **newterm**( ), or **stdin** in the case that **initscr**( ) was used, will be used to do this typeahead checking. The **typeahead**( ) routine specifies that the file descriptor **fd** is to be used to check for typeahead instead. If **fd** is -1, then no typeahead checking will be done.

Environment Queries

baudrate( )

Returns the output speed of the terminal. The number returned is in bits per second, for example 9600, and is an integer.

char erasechar( )

The user's current erase character is returned.

has_ic( )

True if the terminal has insert- and delete-character capabilities.

has_il( )

True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using **scrollok**( ).

char killchar( )

The user's current line kill character is returned.

char *longname( )

This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to **initscr**( ) or **newterm**( ). The area is overwritten by each call to **newterm**( ) and is not restored by **set_term**( ), so the value should be saved between calls to **newterm**( ) if **longname**( ) is going to be used with multiple terminals.

Soft Labels

Curses will manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, *curses* will take over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable LINES. *Curses* standardizes on 8 labels of up to 8 characters each.

slk_init(fmt)
int fmt;

In order to use soft labels, this routine is to be called before **initscr**( ) or **newterm**( ) is called. If **initscr**( ) winds up using a line from

stdscr to emulate the soft labels, then **fmt** determines how the labels are arranged on the screen. Setting **fmt** to **0** indicates that the labels are to be arranged in a 3-2-3 arrangement; **1** asks for a 4-4 arrangement.

int slk_set(labnum, label, fmt)
int labnum;
char *label;
int fmt;

labnum is the label number, from **1** to **8**. **label** is the string to be put on the label, up to **8** characters in length. A NULL string or a NULL pointer will put up a blank label. **fmt** is one of **0**, **1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

slk_refresh()

slk_noutrefresh()

These routines correspond to the routines **wrefresh()** and **wnoutrefresh()**. Most applications would use **slk_noutrefresh()** because a **wrefresh()** will most likely soon follow.

char *slk_label(labnum)
int labnum;

The current label for label number **labnum**, with leading and trailing blanks stripped, is returned.

slk_clear()

The soft labels are cleared from the screen.

slk_restore()

The soft labels are restored to the screen after a **slk_clear()**.

slk_touch()

All of the soft labels are forced to be output the next time a **slk_noutrefresh()** is performed.

Terminfo Level Routines

These low level routines must be called by programs that need to deal directly with the terminfo database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, **setupterm()** should be called. [Note that **setupterm()** is automatically called by **initscr()** and **newterm()**.] This will define the set of terminal-dependent variables defined in TERMINFO(TI_ENV). The *terminfo* variables **lines** and **columns** are initialized by **setupterm**( ) as follows. If the environment variables LINES and COLUMNS exist, their values are used. If the above environment variables do not exist and the program is running in a window, the current window size is used. Otherwise, the values for **lines** and **columns** specified in the *terminfo* database are used.

The header files <**curses.h**> and <**term.h**> should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm**( ) to instantiate them. All *terminfo* strings [including the output of **tparm**( )] should be printed with **tputs**( ) or **putp**( ). Before exiting, **reset_shell_mode**( ) should be called to restore the tty modes. Programs which use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting. (Programs desiring shell escapes should call **reset_shell_mode**( ) and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call **reset_prog_mode**( ) after returning from the shell.)

setupterm(term, fildes, errret)
char *term;
int fildes;
int *errret;

setterm(term)
char *term;

Read in the *terminfo* database, initializing the *terminfo* structures, but do not set up the output virtualization structures used by *curses*. The terminal type is the character string **term;** if **term** is null, the environment variable **TERM** will be used. All output is to file descriptor **fildes.** If **errret** is not NULL, then **setupterm**( ) will return **OK** or **ERR** and store a status value in the integer pointed to by **errret.** A status of 1 in **errret** is normal, 0 means that the terminal could not be found, and -1 means that the terminfo database could not be found. If **errret** is NULL, **setupterm**( ) will print an error message upon finding an error and exit. Thus, the simplest call is **setupterm ((char *) 0, 1, (int *) 0)**, which uses all the defaults.

NOTE: The **setterm**( ) routine is being replaced by **setupterm**( ). The call **setupterm(term**, 1, (int *) 0) provides the same functionality as **setterm(term)**. **setterm**( ) is included here for compatibility and is supported at level 2.

char *tparm(str, p1, p2, ..., p9)
char *str;
int p1, p2, p3, p4, p5, p6, p7, p8, p9;

Instantiate the string **str** with parms **pi**. A pointer is returned to the result of **str** with the parameters applied.

tputs(str, affcnt, putc)
char *str;
int affcnt;
int (*putc)( );

Apply padding information to the string **str** and output it. **str** must be a *terminfo* string variable or the return value from **tparm( )**, **tgetstr( )**, or **tgoto( )**. **affcnt** is the number of lines affected, or 1 if not applicable. **putc( )** is a **putchar** like routine to which the characters are passed, one at a time.

putp(str)
char *str;

A routine that calls **tputs(str, 1, putchar)**.

vidputs(attrs, putc)
int attrs;
int (*putc)( );

Output the string to put the terminal in the video attribute mode **attrs,** which is any combination of the attributes listed below. The characters are passed to the **putchar** like routine **putc.**

vidattr(attrs)
int attrs;

Like **vidputs( )**, except that it outputs through **putchar**.

mvcur(oldrow, oldcol, newrow, newcol)
int oldrow, oldcol, newrow, newcol;

Low level cursor motion.

The following routines return the value of the capability corresponding to the *terminfo* **capname** passed to them, such as **xenl.**

The **capname** for each capability is given in the table column entitled **capname** code in the capabilities section of the *terminfo*(TI_ENV).

tigetflag(capname)
char *capname;

The value -1 is returned if **capname** is not a boolean capability.

tigetnum(capname)
char *capname;
The value -2 is returned if **capname** is not a numeric capability.

tigetstr(capname)
char *capname;
The value (char *) -1 is returned if **capname** is not a string capability.

char *boolnames, *boolcodes, *boolfnames

char *numnames, *numcodes, *numfnames

char *strnames, *strcodes, *strfnames
These null-terminated arrays contain the **capnames**, the *termcap* codes, and the full C names, for each of the *terminfo* variables.

Termcap Compatibility Routines

These routines were included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the **terminfo** database.

tgetent(bp, name)
char *bp, *name;
Look up termcap entry for **name.** The emulation ignores the buffer pointer **bp.**

tgetflag(id)
char id[2];
Get the boolean entry for **id.**

tgetnum(id)
char id[2];
Get numeric entry for **id.**

char *tgetstr(id, area)
char id[2];
char **area;
Return the string entry for **id. tputs()** should be used to output the returned string.

char *tgoto(cap, col, row)
char *cap;
int col, row;

Instantiate the parameters into the given capability. The output from this routine is to be passed to **tputs( )**.

tputs(str,affcnt,putc)
char *str;
int affcnt;
int (*putc)( );

[See **tputs( )** under **Terminfo Level Routines** above.]

Miscellaneous

char *unctrl(c)
chtype c;

This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

NOTE: **unctrl** is a macro, which is defined in <**unctrl.h**>.

char *keyname(c)
int c;

A character string corresponding to the key **c** is returned.

gettmode( )

No-op.

NOTE: **gettmode( )** is included here for compatibility and is supported at level 2.

int scr_dump(filename)
char *filename;

The current contents of the virtual screen are written to the file **filename**.

int scr_restore(filename)
char *filename;

The virtual screen is set to the contents of **filename**, which must have been written using **scr_dump( )**. The next call to **doupdate( )** will restore the screen to what it looked like in the dump file.

int scr_init(filename)
char *filename;

The contents of **filename** are read in and used to initialize the **curses** data structures about what the terminal currently has on its screen. If the data is determined to be valid, **curses** will base its next update of the screen on this information rather than clearing the screen and starting from scratch. **scr_init**( ) would be used after **initscr**( ) or a **system** [see SYSTEM(BA_LIB)] call to share the screen with another process, which has done a **scr_dump**( ) after its **endwin**( ) call. The data will be declared invalid if the time-stamp of the tty is old or the *terminfo* capability **rmcup** exists.

Attributes

The following video attributes, defined in <**curses.h**>, can be passed to the routines **attron**( ), **attroff**( ), and **attrset**( ), or OR'ed with the characters passed to **addch**( ).

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_ALTCHARSET	Alternate character set
A_CHARTEXT	Bit-mask to extract a character
A_ATTRIBUTES	Bit-mask to extract attributes

Function Keys

The following function keys, defined in <**curses.h**>, might be returned by **getch**( ) if **keypad**( ) has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo* database.

Name	*Octal Value*	*Key name*
KEY_BREAK	0401	Break key
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	Backspace
KEY_F0	0410	Function keys; space for 64 keys is reserved.
KEY_F(n)	(KEY_F0+(n))	
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backward (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send
KEY_SRESET	0530	Soft (partial) reset
KEY_RESET	0531	Reset or hard reset
KEY_PRINT	0532	Print or copy

Name	*Octal Value*	*Key name*
KEY_LL	0533	Home down or bottom (lower left) Keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad
KEY_BTAB	0541	Back tab key
KEY_BEG	0542	Beg(inning) key
KEY_CANCEL	0543	Cancel key
KEY_CLOSE	0544	Close key
KEY_COMMAND	0545	Cmd (command) key
KEY_COPY	0546	Copy key
KEY_CREATE	0547	Create key
KEY_END	0550	End key
KEY_EXIT	0551	Exit key
KEY_FIND	0552	Find key
KEY_HELP	0553	Help key
KEY_MARK	0554	Mark key
KEY_MESSAGE	0555	Message key
KEY_MOVE	0556	Move key
KEY_NEXT	0557	Next object key
KEY_OPEN	0560	Open key
KEY_OPTIONS	0561	Options key
KEY_PREVIOUS	0562	Previous object key
KEY_REDO	0563	Redo key
KEY_REFERENCE	0564	Ref(erence) key
KEY_REFRESH	0565	Refresh key
KEY_REPLACE	0566	Replace key
KEY_RESTART	0567	Restart key
KEY_RESUME	0570	Resume key

Name	*Octal Value*	*Key name*
KEY_SAVE	0571	Save key
KEY_SBEG	0572	Shifted beginning key
KEY_SCANCEL	0573	Shifted cancel key
KEY_SCOMMAND	0574	Shifted command key
KEY_SCOPY	0575	Shifted copy key
KEY_SCREATE	0576	Shifted create key
KEY_SDC	0577	Shifted delete char key
KEY_SDL	0600	Shifted delete line key
KEY_SELECT	0601	Select key
KEY_SEND	0602	Shifted end key
KEY_SEOL	0603	Shifted clear line key
KEY_SEXIT	0604	Shifted exit key
KEY_SFIND	0605	Shifted find key
KEY_SHELP	0606	Shifted help key
KEY_SHOME	0607	Shifted home key
KEY_SIC	0610	Shifted input key
KEY_SLEFT	0611	Shifted left arrow key
KEY_SMESSAGE	0612	Shifted message key
KEY_SMOVE	0613	Shifted move key
KEY_SNEXT	0614	Shifted next key
KEY_SOPTIONS	0615	Shifted options key
KEY_SPREVIOUS	0616	Shifted prev key
KEY_SPRINT	0617	Shifted print key
KEY_SREDO	0620	Shifted redo key
KEY_SREPLACE	0621	Shifted replace key
KEY_SRIGHT	0622	Shifted right arrow
KEY_SRSUME	0623	Shifted resume key
KEY_SSAVE	0624	Shifted save key
KEY_SSUSPEND	0625	Shifted suspend key
KEY_SUNDO	0626	Shifted undo key
KEY_SUSPEND	0627	Suspend key
KEY_UNDO	0630	Undo key

LINE GRAPHICS

The following variables may be used to add line-drawing characters to the screen with **waddch()**. When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default character listed below will be stored in the variable. The names were chosen to be consistent with the "VT100" nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left corner
ACS_LLCORNER	+	lower left corner
ACS_URCORNER	+	upper right corner
ACS_LRCORNER	+	lower right corner
ACS_RTEE	+	right tee (┤)
ACS_LTEE	+	left tee (├)
ACS_BTEE	+	bottom tee (┴)
ACS_TTEE	+	top tee (┬)
ACS_HLINE	-	horizontal line
ACS_VLINE	¦	vertical line
ACS_PLUS	+	plus
ACS_S1	-	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

RETURN VALUE

All routines return the integer **ERR** upon failure and an integer value other than **ERR** upon successful completion. Unless otherwise noted in the preceding routine descriptions.

All macros return the value of the **w** version, except **setsrreg**( ), **wsetscrreg**( ), **getyx**( ), **getbegyx**( ), **getmaxyx**( ). The return values of **setsrreg**( ), **wsetscrreg**( ), **getyx**( ), **getbegyx**( ), and **getmaxyx**( ) are undefined (i.e., these should not be used as the right-hand side of assignment statements).

Routines that return pointers always return **(type *) NULL** on error.

SEE ALSO

TERMINFO(TI_ENV).

USAGE

Application Program.

The header file <**curses.h**> automatically includes the header files <**stdio.h**> and <**unctrl.h**>.

LEVEL

Level 1: All routines except *fixterm*( ), *gettmode*( ), *resetterm*( ), *saveterm*( ), *setterm*( ), and the *termcap* compatibility routines *tgetent*( ), *tgetflag*( ), *tgetnum*( ), *tgetstr*( ), and *tgoto*( ).

Level 2: December 1, 1985 for *fixterm*( ), *gettmode*( ), *resetterm*( ), *saveterm*( ), *setterm*( ), and the *termcap* compatibility routines *tgetent*( ), *tgetflag*( ), *tgetnum*( ), *tgetstr*( ), and *tgoto*( ).

Chapter 11
Commands and Utilities

NAME

tic - *terminfo* compiler

SYNOPSIS

tic [–v[n]] [–c] *file*

DESCRIPTION

The command **tic** translates a *terminfo* file from the source format into the compiled format. The results are placed in the directory **/usr/lib/terminfo.** The compiled format is necessary for use with the library routines described in CURSES(TI_LIB). The argument *file* contains one or more *terminfo* terminal descriptions in source format [see TERMINFO(TI_ENV)].

The option **–v** (verbose) causes **tic** to output trace information showing its progress. The optional integer *n* is a number from 1 to 10, inclusive, indicating the desired level of detail of information. If *n* is omitted, the default level is 1. If *n* is specified and greater than 1, the level of detail is increased.

The option **–c** only checks *file* for errors.

The command **tic** compiles all *terminfo* descriptions in the given file. Each description in the file describes the capabilities of a particular terminal. When a **use=***entry_name* field is discovered in the terminal entry currently being compiled, **tic** duplicates the capabilities in *entry_name* for the current entry, with the exception of those capabilities that are explicitly referenced in the current entry.

If the environment variable **TERMINFO** is set, the compiled results are placed there instead of **/usr/lib/terminfo.**

Total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/?/*	Compiled terminal description database

SEE ALSO

CURSES(TI_LIB), TERMINFO(TI_ENV).

USAGE

Administrator.

When an entry, e.g., **entry_name_1**, contains a **use=***entry_name_2* field, any cancelled capabilities in *entry_name_2* must also appear in **entry_name_1** before **use=** for these capabilities to be cancelled in **entry_name_1**.

LEVEL

Level 1.

NAME

tput - initialize a terminal or query the *terminfo* database

SYNOPSIS

tput [**-T***type*] capname [parms ...]

tput [**-T***type*] **init**

tput [**-T***type*] **longname**

tput [**-T***type*] **reset**

DESCRIPTION

The command **tput** uses the *terminfo* database to make the values of terminal-dependent capabilities and information available to the shell [see SH(BU_CMD)], to initialize or reset the terminal, or return the long name of the requested terminal type. The command **tput** outputs a string if the attribute is of type string, or an integer if the attribute is of type integer. If the attribute is of type boolean, **tput** simply sets the exit code (**0** for TRUE if the terminal has the capability, **1** for FALSE if it does not), and produces no output.

-Ttype indicates the type of terminal. Normally this option is unnecessary, as the default is taken from the environment variable **TERM.** If **-T** is specified, then the shell variables **LINES** and **COLUMNS** and the layer size will not be referenced.

capname indicates the attribute from the *terminfo* database. [See TERMINFO(TI_ENV)].

parms If the attribute is a string that takes parameters, the arguments **parms** will be instantiated into the string. An all numeric argument will be passed to the attribute as a number.

init If the *terminfo* database is present and an entry for the user's terminal exists, then the following will occur: (1) if present, the terminal's initialization strings will be output (**is1, is2, is3, if, iprog**) (2) any delays (e.g., newline) specified in the entry will be set in the tty driver (3) tabs expansion will be turned on or off according to the specification in the entry, and (4) if tabs are not expanded, standard tabs will be set (every 8 spaces). If an entry does not contain the information needed for any of the four above activities, that activity will silently be skipped.

longname If the *terminfo* database is present and an entry for the user's terminal exists, then the long name of the terminal will be output. The long name is the last name in the first line of the terminal's description in the *terminfo* database.

reset **reset** behaves identically like **init** with the following exception. Instead of outputting initialization strings, the terminal's reset strings will be output if present (**rs1, rs2, rs3, rf**). If the reset strings are not present, but initialization strings are, the initialization strings will be output.

EXAMPLES

tput clear
Echo clear-screen sequence for the current terminal.

tput cols
Print the number of columns for the current terminal.

tput -T450 cols
Print the number of columns for the 450 terminal.

bold=\`tput smso\`
offbold=\`tput rmso\`
Set the shell variables "bold" to begin standout mode sequence and "offbold" to end standout mode sequence for the current terminal. This might be followed by a prompt, e.g.:
echo "${bold}Name: ${offbold}\c"

tput hc
Set exit code to indicate if the current terminal is a hardcopy terminal.

tput cup 23 4
Print the sequence to move the cursor to row 23, column 4.

tput longname
Print the long name from the *terminfo* database for the type of terminal specified in the environmental variable **TERM**.

tput init
Initialize the terminal according to the type of terminal in the environmental variable **TERM**. This command should be included in everyone's .profile after the environmental variable **TERM** has been exported.

tput -T5620 reset
Reset an AT&T 5620 terminal, overriding the type of terminal in the environmental variable **TERM**.

tput cup 0 0
Send the sequence to move the cursor to row **0**, column **0** (the upper left corner of the screen, usually known as the "home" cursor position).

FILES

/usr/lib/terminfo/?/*	Compiled terminal description database *terminfo*(TI_ENV).

RETURN VALUE

If **capname** is of type boolean, a value of **0** is returned for TRUE and **1** for FALSE.

If **capname** is of type string, a value of **0** is returned; if the **capname** is defined for this terminal *type* (the value of **capname** is returned on standard output); a value of **1** is returned if **capname** is not defined for this terminal *type* (a null value is returned on standard output).

If **capname** is of type integer, a value of **0** is returned if **capname** is defined for this terminal *type*.

The following error codes are returned:

2 usage error

3 unknown terminal *type* or no *terminfo* database

4 unknown *terminfo* capability **capname**
(i.e., *terminfo* does not support a capability named **capname**).

SEE ALSO

STTY(BU_CMD), TERMINFO(TI_ENV).

USAGE

Application Program.

tput init or **tput reset** may clear the user's screen.

LEVEL

Level 1.

Part IV

Network Services Extension Definition

Chapter 12 Introduction

12.1 INTRODUCTION

The NETWORK SERVICES EXTENSION provides advanced standard interfaces to support networking applications. It is divided into three functional areas: OPEN SYSTEMS NETWORKING INTERFACES, STREAMS I/O INTERFACES, and the SHARED RESOURCE ENVIRONMENT. Consistent with the definition of Conforming Systems (see section 1.2.2), a conforming system must support all components defined for each of these three functional areas.

The OPEN SYSTEMS NETWORKING INTERFACES section describes functions that provide a protocol independent application interface to networking services based on the service definitions of the OSI (Open Systems Interconnection) Reference Model. Application developers access the functions that provide services at a particular level and need not care about the protocol implementation that is providing those services. The functions defined at this time provide the services of the OSI Transport Layer. These services provide end-to-end data transmission using the services of an underlying network. Applications written using the transport interface are independent of the underlying protocols. By providing media and protocol independence, the interface enables networking applications to have the flexibility to run in various protocol environments.

The STREAMS I/O INTERFACES section describes the interfaces that enable a user to directly access protocol modules that are implemented in the kernel using the streams framework. Streams provides a uniform mechanism for implementing network services in the kernel by defining standard interfaces for device drivers and protocol modules.

The SHARED RESOURCE ENVIRONMENT section describes new capabilities for sharing and administering resources among interconnected machines. These new capabilities are collectively known as *Remote File Sharing.* Using *Remote File Sharing,* files that physically reside on a remote machine can be accessed as if they were on the local machine; the capabilities described here provide the interface for accessing and managing *Remote File Sharing.* New utilities provide the basic functionality, while additional functionality is added to the Base System, the BASIC UTILITIES EXTENSION, and the ADMINISTERED SYSTEM EXTENSION.

The components of the NETWORK SERVICES EXTENSION are new in System V Release 3. The NETWORK SERVICES EXTENSION is dependent upon the Base System as defined for System V Release 3.

Chapter 13
Open Systems Networking Interfaces

13.1 INTRODUCTION

The OPEN SYSTEMS NETWORKING INTERFACES section of the NETWORK SERVICES EXTENSION describes functions that provide a protocol independent application interface to networking services based on the service definitions of the OSI (Open Systems Interconnection) Reference Model. Application developers access the functions that provide services at a particular level and need not care about the protocol implementation that is providing those services.

The functions defined at this time provide the services of the OSI Transport Layer. These services provide end-to-end data transmission using the services of an underlying network. Applications written using the transport interface are independent of the underlying protocols. By providing media and protocol independence, the interface enables networking applications to have the flexibility to run in various protocol environments.

This section of the extension is dependent upon the Base System.

13.2 FUTURE DIRECTIONS

As interfaces to other layers of the OSI Reference Model become defined for System V, the functions providing services of these layers will be included in the OPEN SYSTEMS NETWORKING INTERFACES library.

13.3 DESCRIPTION

LIBRARY ROUTINES

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

HEADER FILES

tiuser.h

ERROR CONDITIONS

EPROTO **Protocol Error**

13.4 DEFINITIONS

Transport user

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

Active transport user

The transport user that initiates a connection.

Passive transport user

The transport user that listens for an incoming connect indication.

Transport provider

The transport protocol that provides the services of the transport interface.

Transport endpoint

The communication path, which is identified by a file descriptor, between a transport user and a specific transport provider.

Protocol address

The address, also known as the Transport Service Access Point (TSAP) address, that identifies the transport user. This interface places no structure or semantics on an address.

Connection mode

A circuit-oriented mode of transfer in which data is passed from one user to another over an established connection in a reliable, sequenced fashion.

Connectionless mode

A mode of transfer in which data is passed from one user to another in self-contained units with no logical relationship required among multiple units.

Synchronous execution

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

Asynchronous execution

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

TSDU

The Transport Service Data Unit, 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).

ETSDU

The Expedited Transport Service Data Unit, which 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).

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 */
```

13.5 EFFECTS ON THE BASE SYSTEM

Components in the Base System may return a new value for **errno** as listed below. An application that checks the value of **errno** must include the header file <**errno.h**>.

The following symbolic name defines an additional error return condition:

Name	Description
EPROTO	Protocol Error

13.6 EFFECTS ON THE SOFTWARE DEVELOPMENT EXTENSION

In a software development environment, a program **file.c** that accesses any function defined in this part of the extension must be compiled in one of the following ways:

cc file.c –lnsl_s

or

cc file.c –lnsl

13.7 TRANSPORT SERVICE INTERFACE

The Open Systems Networking Interfaces provide the services of strategic levels of the Open Systems Interconnection (OSI) Reference Model [1]. The services currently defined in this library conform to those services specified in the ISO Transport Service Definition document [2] for both connection-mode and connectionless-mode transport services. Functions to support services of other layers of the OSI Reference Model will be added to this library as deemed necessary.

13.7.1 Overview

A set of functions has been defined to provide a transport service interface for user processes and to be independent of any specific transport protocol. This transport service enables two user processes to transfer data between them over a communications channel.

In order to properly use the library functions that are defined, certain rules must be followed. This overview is intended to describe the relationship among the functions and show how a developer would write an application using these functions. State tables are included to show the allowable sequences of function calls given a particular state and event.

The remainder of this interface description refers to the concept of a *transport endpoint*. This endpoint specifies a communications path between a transport user and a specific transport provider, and is identified by a local file descriptor (**fd**). In other words, a transport endpoint is manifested as an open device special file. A transport provider is defined to be the transport protocol that provides the services of the transport layer. All requests to the transport provider must pass through a transport endpoint. The file descriptor **fd** is returned by the function T_OPEN(NS_LIB) and is used as an argument to subsequent functions to identify the transport endpoint.

Modes of Service

The transport service interface supports two modes of service: connection mode and connectionless mode. A single transport endpoint may not support both modes of service simultaneously.

The connection-mode transport service is circuit-oriented and enables data to be transferred over an established connection in a reliable, sequenced manner. This service enables the negotiation of the parameters and options that govern the transfer of data. It provides an identification mechanism that avoids the overhead of address transmission and resolution during the data transfer phase. It also provides a context in which successive units of data, transferred between peer users, are logically related. This service is attractive to applications that require relatively long-lived, datastream-oriented interactions.

In contrast, the connectionless-mode transport service is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. These units are also known as datagrams. This service requires only a preexisting association between the peer users involved, which determines the characteristics of the data to be transmitted. No dynamic negotiation of parameters and options is supported by this service. All the information required to deliver a unit of data (e.g., destination address) is presented to the transport provider, together with the data to be transmitted, in a single service access which need not relate to any other service access. Also, each unit of data transmitted is entirely self-contained, and can be independently routed by the transport provider. This service is attractive to applications that involve short-term request/response interactions, exhibit a high level of redundancy, are dynamically reconfigurable, or do not require guaranteed, in-sequence delivery of data.

Error Handling

Two levels of error are defined for the transport interface. The first is the library error level. Each library function has one or more error returns. Failures are indicated by a return value of -1. An external integer, **t_errno**, holds the specific error number when such a failure occurs. This value is set when errors occur but is not cleared on successful library calls, so it should be tested only after an error has been indicated. A diagnostic function, T_ERROR(NS_LIB), is provided for printing out information on the current transport error. The state of the transport provider may change if a transport error occurs.

The second level of error is the operating system service routine level. A special library level error number has been defined called **TSYSERR** which is generated by each library function when an operating system service routine fails or some general error occurs. When a function sets **t_errno** to **TSYSERR**, the specific system error may be accessed through the external variable **errno**.

A new system error, **EPROTO**, has been defined to support System V networking. This error is generated by the transport provider when a protocol error has occurred. If the error is severe, it may cause the file descriptor and transport endpoint to be unusable. To continue in this case, all users of the file must close it. Then the file may be re-opened and initialized.

Synchronous and Asynchronous Execution Modes

The transport service interface is inherently asynchronous; various events may occur independent of the actions of a transport user. For example, a user may be sending data over a transport connection when an asynchronous disconnect indication arrives. The user must somehow be informed that the connection has been broken.

The transport service interface supports two execution modes for handling asynchronous events: synchronous mode and asynchronous mode. In the synchronous mode of operation, the transport functions wait for specific events before returning control to the user. While waiting, the user cannot perform other tasks. For example, a function that attempts to receive data in synchronous mode will wait until data arrives before returning control to the user. This is the default mode of execution. It is useful for user processes that want to wait for events to occur, or for user processes that have no other useful work to perform.

The asynchronous mode of operation, on the other hand, provides a mechanism for notifying a user of some event without forcing the user to wait for that event. The handling of networking events in an asynchronous manner is seen as a desirable capability of the transport interface. This would enable users to perform useful work while waiting for a particular event. For example, a function that attempts to receive data in asynchronous mode will return control to the user immediately if no data is available. The user may then periodically poll for incoming data until it arrives. The asynchronous mode is intended for those applications that expect long delays between events and have other tasks that they can perform in the meantime.

The two execution modes are not provided through separate interfaces or different functions. Instead, functions that process incoming events have two modes of operation: synchronous and asynchronous. The desired mode is specified through the **O_NDELAY** flag, which may be set when the transport provider is initially opened, or before any specific function or group of functions is executed using the FCNTL(BA_OS) operating system service routine. The effect of this flag is completely specified in the description of each function.

Eight asynchronous events are defined in the transport service interface to cover both connection-mode and connectionless-mode service. They are represented as separate bits in a bitmask using the following defined symbolic names:

T_LISTEN	This event occurs when a connect request from a remote user is received by a transport provider (connection-mode service only).
T_CONNECT	This event occurs when a connect confirmation is received by a transport provider (connection-mode service only).
T_DATA	This event occurs when normal data is received by a transport provider.
T_EXDATA	This event occurs when expedited data is received by a transport provider (connection-mode service only).
T_DISCONNECT	This event occurs when a disconnect indication is received by a transport provider (connection-mode service only).
T_ORDREL	This event occurs when an orderly release indication is received by a transport provider (connection-mode service with orderly release only).
T_ERROR	This event occurs when a fatal error is generated by the transport provider, thus making the transport endpoint inaccessible.
T_UDERR	This event occurs when an error is found in a previously sent data unit (connectionless-mode service only).

A process that issues functions in synchronous mode must still be able to recognize certain asynchronous events immediately and act on them if necessary. This is handled through a special transport error **TLOOK** which is returned by a function when an asynchronous event occurs. The T_LOOK(NS_LIB) function is then invoked to identify the specific event that has occurred when this error is returned.

Asynchronous processing is accomplished through polling. The polling capability enables processes to do useful work and periodically poll for one of the above asynchronous events. This facility is provided by setting **O_NDELAY** for the appropriate function(s) and by using the T_LOOK(NS_LIB) function to do the polling.

13.7.2 Overview of the Connection-mode Service

The connection-mode transport service consists of four phases of communication: initialization/de-initialization, connection establishment, data transfer, and connection release. A state machine is described in the section *Transport Service Interface Sequence of Functions* and Figure 13-8 that defines the legal sequence in which functions from each phase may be issued.

Initialization/De-initialization Phase

Before a user can attempt to establish a transport connection, the environment of the user must be initialized. Specifically, the user must create a local communication path to the transport provider (i.e., create the transport endpoint), obtain necessary protocol-specific information, and activate the transport endpoint. A transport endpoint is viewed as active when the transport provider may accept or request connections associated with the endpoint.

After a connection has been released, the transport user must de-initialize the associated transport endpoint, thereby freeing the resource for future use.

The functions that support initialization/de-initialization tasks are described below. All such functions provide local management functions; no information is sent over the network.

T_OPEN(NS_LIB) — This function creates a transport endpoint and returns protocol-specific information associated with that endpoint. It also returns a file descriptor that serves as the local identifier of the endpoint.

T_BIND(NS_LIB) — This function associates a protocol address with a given transport endpoint, thereby activating the endpoint. It also directs the transport provider to begin accepting connect indications if so desired.

T_OPTMGMT(NS_LIB) — This function enables the user to get or negotiate protocol options with the transport provider.

T_UNBIND(NS_LIB) — This function disables a transport endpoint such that no further request destined for the given endpoint will be accepted by the transport provider.

T_CLOSE(NS_LIB) — This function informs the transport provider that the user is finished with the transport endpoint, and frees any local resources associated with that endpoint.

The following functions are also local management functions, but can be issued during any phase of communication.

T_GETINFO(NS_LIB) — This function returns protocol-specific information associated with the specified transport endpoint.

T_GETSTATE(NS_LIB) — This function returns the current state of the transport endpoint.

T_SYNC(NS_LIB) — This function synchronizes the data structures managed by the transport library with the transport provider.

T_ALLOC(NS_LIB) — This function allocates storage for the specified library data structure.

T_FREE(NS_LIB) — This function frees storage for a library data structure that was allocated by T_ALLOC(NS_LIB).

T_ERROR(NS_LIB) — This function prints out a message describing the last error encountered during a call to a transport library function.

T_LOOK(NS_LIB) — This function returns the current event associated with the given transport endpoint.

Connection Establishment Phase

This phase enables two transport users to establish a transport connection between them. In the connection establishment scenario, one user is considered active and initiates the conversation, while the second user is passive and waits for a transport user to request a connection.

The active user requests a connection and then receives a response from the called user. The passive user waits for connect indications (i.e., indications of a connect request) and then either accepts or rejects the request. The functions that support these operations are:

T_CONNECT(NS_LIB) — This function requests a connection to the transport user at a specified destination, and waits for the remote user's response. This function may be executed in either synchronous or asynchronous mode. In synchronous mode, the function waits for the remote user's response before returning control to the local user. In asynchronous mode, the function initiates connection establishment but returns control to the local user before a response arrives.

T_RCVCONNECT(NS_LIB) — This function enables an active transport user to determine the status of a previously sent connect request. If the request was accepted, the connection establishment phase will be complete on return from this function. This function is used in conjunction with T_CONNECT(NS_LIB) to establish a connection in an asynchronous manner.

T_LISTEN(NS_LIB) — This function enables the passive transport user to receive connect indications from other transport users.

T_ACCEPT(NS_LIB) — This function is issued by the passive user to accept a particular connect request after an indication has been received.

Data Transfer Phase

Once a transport connection has been established between two users, data may be transferred back and forth over the connection. Two functions have been defined to support data transfer in connection mode as follows:

T_SND(NS_LIB) — This function enables transport users to send either normal or expedited data over a transport connection.

T_RCV(NS_LIB) — This function enables transport users to receive either normal or expedited data on a transport connection.

Connection Release Phase

Two forms of connection release are supported in the connection-mode transport interface: abortive and orderly. An abortive release may be invoked from either the connection establishment phase or the data transfer phase. When in the connection establishment phase, a transport user may use the abortive release to reject a connect request. In the data transfer phase, either user may abort a connection at any time. The abortive release is not negotiated by the transport users and it takes effect immediately on request. The user on the other side of the connection is notified when a connection is aborted. The transport provider may also initiate an abortive release, in which case both users are informed that the connection no longer exists. There is no guarantee of delivery of user data once an abortive release has been initiated.

The orderly release capability is an optional feature of the connection-mode service. If supported by the underlying transport provider, orderly release may be invoked from the data transfer phase to enable two users to gracefully release a connection. The procedure for orderly release prevents the loss of data that may occur during an abortive release.

The functions that support connection release are:

T_SNDDIS(NS_LIB) This function can be issued by either transport user to initiate the abortive release of a transport connection. It may also be used to reject a connect request during the connection establishment phase.

T_RCVDIS(NS_LIB) This function identifies the reason for the abortive release of a connection, where the connection is released by the transport provider or another transport user.

T_SNDREL(NS_LIB) (Optional). This function can be issued by either transport user to initiate an orderly release. The connection remains intact until both users issue this function and T_RCVREL(NS_LIB).

T_RCVREL(NS_LIB) (Optional). This function is issued when a user is notified of an orderly release request, as a means of informing the transport provider that the user is aware of the remote user's actions.

13.7.3 Overview of the Connectionless-mode Service

The connectionless-mode transport service consists of two phases of communication: initialization/de-initialization and data transfer. A brief description of each phase and its associated functions is presented below. A state machine is described in the section *Transport Service Interface Sequence of Functions* and Figure 13-7 that defines the legal sequence in which functions from each phase may be issued.

Initialization/De-initialization Phase

Before a user can attempt to transfer data in connectionless mode, the environment of the user must be initialized. Specifically, the user must create a local communication path to the transport provider (i.e., create the transport endpoint), obtain necessary protocol-specific information, and activate the transport endpoint. A transport connection endpoint is viewed as active when a transport user may send or receive data units through that endpoint.

When a transport user no longer wishes to send or receive data units through a given transport endpoint, they must de-initialize the endpoint, thereby freeing the resource for future use.

The functions that support the initialization/de-initialization tasks are the same functions used in the connection-mode service.

Data Transfer Phase

Once a transport endpoint has been activated, a user is free to send and receive data units through that endpoint in connectionless mode as follows:

T_SNDUDATA	This function enables transport users to send a self-contained data unit to the user at the specified protocol address.
T_RCVUDATA	This function enables transport users to receive data units from other users.
T_RCVUDERR	This function enables transport users to retrieve error information associated with a previously sent data unit.

13.7.4 Transport Service Interface Sequence of Functions

Figures 13-2 through 13-8 are included to describe the possible states of the transport provider as seen by the transport user, describe the incoming and outgoing events that may occur on any connection, and identify the allowable sequence of function calls. Given a current state and event, the transition to the next state is shown as well as any actions that must be taken by the transport user.

The allowable sequence of functions is described in Figures 13-6, 13-7, and 13-8. The support functions, T_GETSTATE(NS_LIB), T_GETINFO(NS_LIB), T_ALLOC(NS_LIB), T_FREE(NS_LIB), T_LOOK(NS_LIB), and T_SYNC(NS_LIB) are excluded from the state tables because they do not affect the state of the interface. Each of these functions may be issued from any state except the uninitialized state. Similarly, the T_ERROR(NS_LIB) function has been excluded from the state table because it does not affect the state of the interface.

The following are rules regarding the maintenance of the state of the interface.

- It is the responsibility of the transport provider to keep record of the state of the interface as seen by the transport user.
- The transport provider must never process a function that places the interface out of state.
- If the user issues a function out of sequence, the transport provider should indicate this where possible through an error return on that function. The state should not change. In this case, if any data is passed with the function when not in the T_DATAXFER state, that data will not be accepted or forwarded by the transport provider.
- The uninitialized state (T_UNINIT) of a transport endpoint is the initial state, and the endpoint must be initialized and bound before the transport provider may view it as active.

- The uninitialized state is also the final state, and the transport endpoint must be viewed as unused by the transport provider. The T_CLOSE(NS_LIB) function will close the transport provider and free the transport library resources for another endpoint.

- According to the state table in Figure 13-6, T_CLOSE(NS_LIB) should only be issued from the T_UNBND state. If it is issued from any other state and no other user has that endpoint open, the action will be abortive, the transport endpoint will be successfully closed, and the library resources will be freed for another endpoint. When T_CLOSE(NS_LIB) is issued, the transport provider must ensure that the address associated with the specified transport endpoint has been unbound from that endpoint. Also, the provider should send appropriate disconnects if T_CLOSE(NS_LIB) is not issued from the unbound state.

The following rules apply only to the connection-mode transport service:

- The transport connection release phase can be initiated at any time during the connection establishment phase or data transfer phase.

- The only time the state of a transport service interface of a transport endpoint may be transferred to another transport endpoint is when the T_ACCEPT(NS_LIB) function specifies such action. The following rules then apply to the cooperating transport endpoints:

  — The endpoint that is to accept the current state of the interface must be bound to an appropriate protocol address and must be in the T_IDLE state.

  — The user transferring the current state of an endpoint must have correct permissions for the use of the protocol address bound to the accepting transport endpoint.

  — The endpoint that transfers the state of the transport interface is placed into the T_IDLE state by the transport provider after the completion of the transfer if there are no more outstanding connect indications.

13.7.5 Guidelines for Writing Protocol-Independent Software

A primary goal of the user-level transport interface is that it be independent of any particular transport protocol. More importantly, the interface was designed to enable users to write programs that had no knowledge of the particular transport protocol to which they would interface. This will enable networking applications to be run in different protocol environments without change.

The user-level transport interface will support protocol-independence for applications if the following guidelines are followed:

1. In the connection-mode service, the concept of a transport service data unit (TSDU) may not be supported by all transport providers. The user should make no assumptions about the preservation of logical data boundaries across a connection.

2. The protocol-specific service limits returned on the T_OPEN(NS_LIB) and T_GETINFO(NS_LIB) functions must not be exceeded. It is the responsibility of the user to access these limits and then adhere to the limits throughout the communication process.

3. The user program should not look at or change options that are specific to the underlying protocol. The T_OPTMGMT(NS_LIB) function enables a user to access default protocol options from the transport provider, which may then be blindly passed as an argument on the appropriate connect establishment function. Optionally, the user can choose not to pass options as an argument on connect establishment functions.

4. Protocol-specific addressing issues should be hidden from the user program. The user program should not specify any protocol address on the T_BIND(NS_LIB) function, but instead should allow T_BIND(NS_LIB) to assign an address to the user. In this way, details concerning protocol-specific addressing are hidden from the user.

   Similarly, the user must have some way of accessing destination addresses in an invisible manner, such as through a name server. However, the details for doing so are outside the scope of this interface specification.

5. The reason codes associated with T_RCVDIS(NS_LIB) are protocol-dependent. The user should not interpret this information if protocol-independence is a concern.

6. The error codes associated with T_RCVUDERR(NS_LIB) are protocol-dependent. The user should not interpret this information if protocol-independence is a concern.

7. The names of devices should not be hard-coded into programs. While software may be written for a particular class of service (e.g., connectionless-mode service), it should not be written to depend on any attribute of the underlying protocol.

8. The optional orderly release facility of the connection-mode service [i.e., T_SNDREL(NS_LIB) and T_RCVREL(NS_LIB)] should not be used by programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols. In particular, its use will prevent programs from successfully communicating with ISO open systems.

13.7.6 Example

The following example (Figure 13-1) shows the allowable sequence of functions of an active user and passive user communicating using a connection-mode transport service. This example is not meant to show all the functions that must be called but rather to highlight the important functions that request a particular service. Blank lines are used to indicate that a function would be issued by one user prior to the issuance of a related function by the remote user. For example, the active user issues T_CONNECT(NS_LIB) to request a connection and the passive user would receive an indication of the connect request [via the return from T_LISTEN(NS_LIB)] and then would issue the T_ACCEPT(NS_LIB).

The state diagram that follows shows the flow of the events through the various states. The active user is represented by a solid line and the passive user is represented by a dashed line. This example shows a successful connection being established and terminated using connection-mode transport service without orderly release. For a detailed description of all possible states and events, see Figure 13-8.

Active User	**Passive User**
t_open	**t_open**
t_bind	**t_bind**
	t_listen
t_connect	
	t_accept
t_rcvconnect	
t_snd	
	t_rcv
t_snddis	
	t_rcvdis
t_unbind	**t_unbind**
t_close	**t_close**

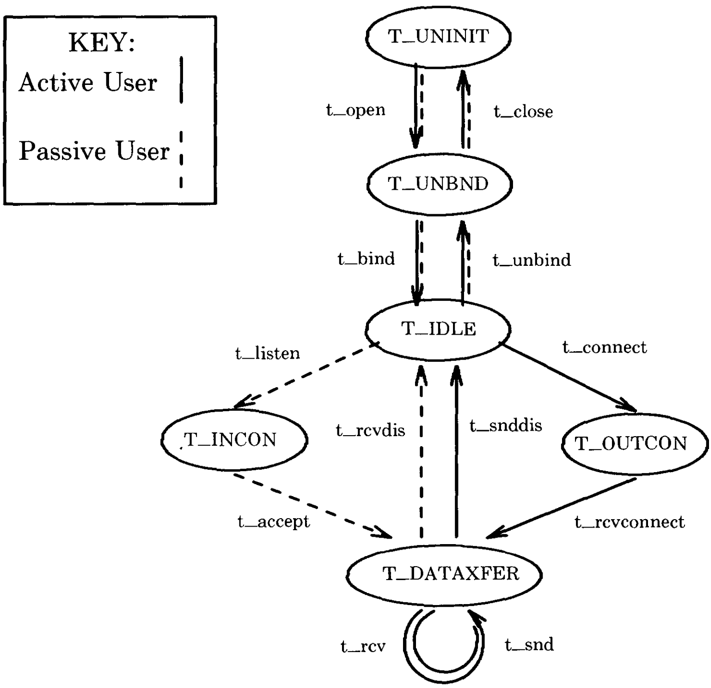

Figure 13-1: Example of a Sequence of Transport Functions

Transport Interface States

The following table (Figure 13-2) describes all possible states of the transport provider as seen by the transport user. The service type may be connection-mode, connection-mode with orderly release, or connectionless-mode.

State	Description	Service Type
T_UNINIT	uninitialized - initial and final state of interface	T_COTS T_CLTS T_COTS_ORD
T_UNBND	unbound	T_COTS T_COTS_ORD T_CLTS
T_IDLE	no connection established	T_COTS T_COTS_ORD T_CLTS
T_OUTCON	outgoing connection pending for active user	T_COTS T_COTS_ORD
T_INCON	incoming connection pending for passive user	T_COTS T_COTS_ORD
T_DATAXFER	data transfer	T_COTS T_COTS_ORD
T_OUTREL	outgoing orderly release (waiting for orderly release indication)	T_COTS_ORD
T_INREL	incoming orderly release (waiting to send orderly release request)	T_COTS_ORD

Figure 13-2: Transport Interface States

Outgoing Events

The following outgoing events correspond to the successful return of the specified user-level transport functions, where these functions send a request or response to the transport provider.

In Figure 13-3, some events (e.g., **acceptX**) are distinguished by the context in which they occur. The context is based on the values of the following:

ocnt count of outstanding connect indications

fd file descriptor of the current transport endpoint

resfd file descriptor of the transport endpoint where a connection will be accepted

Event	Description	Service Type
opened	successful return of **t_open**	T_COTS, T_COTS_ORD, T_CLTS
bind	successful return of **t_bind**	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	successful return of **t_optmgmt**	T_COTS, T_COTS_ORD, T_CLTS
unbind	successful return of **t_unbind**	T_COTS, T_COTS_ORD, T_CLTS
closed	successful return of **t_close**	T_COTS, T_COTS_ORD, T_CLTS
connect1	successful return of **t_connect** in synchronous mode	T_COTS, T_COTS_ORD
connect2	**TNODATA** error on **t_connect** in asynchronous mode, or **TLOOK** error due to a disconnect indication arriving on the transport endpoint.	T_COTS, T_COTS_ORD
accept1	successful return of **t_accept** with **ocnt** == 1, **fd** == **resfd**	T_COTS, T_COTS_ORD
accept2	successful return of **t_accept** with **ocnt** == 1, **fd** != **resfd**	T_COTS, T_COTS_ORD
accept3	successful return of **t_accept** with **ocnt** > 1	T_COTS, T_COTS_ORD
snd	successful return of **t_snd**	T_COTS, T_COTS_ORD
snddis1	successful return of **t_snddis** with **ocnt** <= 1	T_COTS, T_COTS_ORD
snddis2	successful return of **t_snddis** with **ocnt** > 1	T_COTS, T_COTS_ORD
sndrel	successful return of **t_sndrel**	T_COTS_ORD
sndudata	successful return of **t_sndudata**	T_CLTS

Figure 13-3: Transport Interface Outgoing Events

Incoming Events

The following incoming events correspond to the successful return of the specified user-level transport functions, where these functions retrieve data or event information from the transport provider. The only incoming event not associated directly with the return of a function on a given transport endpoint is **pass_conn**, which occurs when a user transfers a connection to another transport endpoint. This event occurs on the endpoint that is being passed the connection, despite the fact that no function is issued on that endpoint. **Pass_conn** is included in the state tables to describe what happens when a user accepts a connection on another transport endpoint.

In Figure 13-4, the **rcvdis** events are distinguished by the context in which they occur. The context is based on the value of **ocnt**, which is the count of outstanding connect indications on the current transport endpoint.

Incoming Event	Description	Service Type
listen	successful return of **t_listen**	T_COTS T_COTS_ORD
rcvconnect	successful return of **t_rcvconnect**	T_COTS T_COTS_ORD
rcv	successful return of **t_rcv**	T_COTS T_COTS_ORD
rcvdis1	successful return of **t_rcvdis** with **ocnt** <= 0	T_COTS T_COTS_ORD
rcvdis2	successful return of **t_rcvdis** with **ocnt** == 1	T_COTS T_COTS_ORD
rcvdis3	successful return of **t_rcvdis** with **ocnt** > 1	T_COTS T_COTS_ORD
rcvrel	successful return of **t_rcvrel**	T_COTS_ORD
rcvudata	successful return of **t_rcvudata**	T_CLTS
rcvuderr	successful return of **t_rcvuderr**	T_CLTS
pass_conn	receive a passed connection	T_COTS T_COTS_ORD

Figure 13-4: Transport Interface Incoming Events

Transport User Actions

Some state transitions are accompanied by a list of actions the transport user must take. These actions are represented by the notation [n], where n is the number of the specific action as described in Figure 13-5.

[1] Set the count of outstanding connect indications to zero.

[2] Increment the count of outstanding connect indications.

[3] Decrement the count of outstanding connect indications.

[4] Pass a connection to another transport endpoint as indicated in T_ACCEPT(NS_LIB).

Figure 13-5: Transport Interface User Actions

State Tables

Figures 13-6 and 13-7 describe the possible next states, given the current state and event. The state is that of the transport provider as seen by the transport user.

The contents of each box represent the next state given the current state (column) and the current incoming or outgoing event (row). An empty box represents a state/event combination that is invalid. Along with the next state, each box may include an action list (as specified in Figure 13-5). The transport user must take the specific actions in the order specified in the state table.

A separate table is shown for initialization/de-initialization, data transfer in connectionless mode, and connection/release/data-transfer in connection mode.

state / event	T_UNINIT	T_UNBND	T_IDLE
opened	T_UNBND		
bind		T_IDLE [1]	
optmgmt			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

Figure 13-6: Initialization/De-initialization State Table

state / event	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Figure 13-7: Data-Transfer State Table for Connectionless-mode Service

state / event	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
connect1	T_DATAXFER					
connect2	T_OUTCON					
rcvconnect		T_DATAXFER				
listen	T_INCON [2]		T_INCON [2]			
accept1			T_DATAXFER [3]			
accept2			T_IDLE [3][4]			
accept3			T_INCON [3][4]			
snd				T_DATAXFER		T_INREL
rcv				T_DATAXFER	T_OUTREL	
snddis1		T_IDLE	T_IDLE [3]	T_IDLE	T_IDLE	T_IDLE
snddis2			T_INCON [3]			
rcvdis1		T_IDLE		T_IDLE	T_IDLE	T_IDLE
rcvdis2			T_IDLE [3]			
rcvdis3			T_INCON [3]			
sndrel				T_OUTREL		T_IDLE
rcvrel				T_INREL	T_IDLE	
pass_conn	T_DATAXFER					

Figure 13-8: Connection/Release/Data-Transfer State Table for Connection-mode Service

REFERENCES

1. CCITT Recommendation X.200 - "Reference Model of Open Systems Interconnection for CCITT Applications", 1984.

2. ISO IS 8072 - "Information Processing Systems - Open Systems Interconnection - Transport Service Definition", 1984.

NAME

t_accept - accept a connect request

SYNOPSIS

#include <tiuser.h>

int t_accept(fd, resfd, call)
int fd;
int resfd;
struct t_call *call;

DESCRIPTION

This function is issued by a transport user to accept a connect request. **Fd** identifies the local transport endpoint where the connect indication arrived, **resfd** specifies the local transport endpoint where the connection is to be established, and **call** contains information required by the transport provider to complete the connection. **Call** points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

In **call**, **addr** is the address of the caller, **opt** indicates any protocol-specific parameters 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(NS_LIB) that uniquely associates the response with a previously received connect indication.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. If the same endpoint is specified (i.e., **resfd=fd**), the connection can be accepted unless the following condition is true: The user has received other indications on that endpoint but has not responded to them [with **t_accept** or T_SNDDIS(NS_LIB)]. For this condition, **t_accept** will fail and set **t_errno** to **TBADF**.

If a different transport endpoint is specified (**resfd!=fd**), the endpoint must be bound to a protocol address and must be in the T_IDLE state [see T_GETSTATE(NS_LIB)] before the **t_accept** is issued.

For both types of endpoints, **t_accept** will fail and set **t_errno** to **TLOOK** if there are indications (e.g., a connect or disconnect) waiting to be received on that endpoint.

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(NS_LIB) or T_GETINFO(NS_LIB). If the **len** field of **udata** is zero, no data will be sent to the caller.

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF]	The file descriptor **fd** or **resfd** does not refer to a transport endpoint, or the user is illegally 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 appropriate 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 illegal information.
[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider.
[TBADSEQ]	An invalid sequence number was specified.
[TLOOK]	An asynchronous event has occurred on the transport endpoint referenced 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.

RETURN VALUE

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.

SEE ALSO

T_CONNECT(NS_LIB), T_GETSTATE(NS_LIB), T_LISTEN(NS_LIB), T_OPEN(NS_LIB), T_RCVCONNECT(NS_LIB).

NAME

t_alloc - allocate a library structure

SYNOPSIS

```
#include <tiuser.h>

char *t_alloc(fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

DESCRIPTION

The **t_alloc** function 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 must be one of the following:

```
T_BIND        struct t_bind
T_CALL        struct t_call
T_OPTMGMT     struct t_optmgmt
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.

Each of the above structures, except **T_INFO**, contains at least one field of type "**struct netbuf**". For each field of this type, the user may specify that the buffer for that field should be allocated as well. The length of the buffer allocated will be based on the size information returned in the **info** argument of T_OPEN(NS_LIB) or T_GETINFO(NS_LIB). The relevant fields of the **info** argument are described in the following list. The **fields** argument specifies which buffers to allocate, 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 (size obtained from **info_addr**).

T_OPT The **opt** field of the **t_optmgmt**, **t_call**, **t_unitdata**, or **t_uderr** structures (size obtained from **info_options**).

T_UDATA The **udata** field of the **t_call**, **t_discon**, or **t_unitdata** structures (for **T_CALL**, size is the maximum value of **info_connect** and **info_discon**; for **T_DIS**, size is the value of **info_discon**; for **T_UNITDATA**, size is the value of **info_tsdu**)..

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, and initialize the **len** field to zero and the **buf** pointer and **maxlen** field accordingly. Because the length of the buffer allocated will be based on the same size information that is returned to the user on T_OPEN(NS_LIB) and T_GETINFO(NS_LIB), **fd** must refer to the transport endpoint through which the newly allocated structure will be passed. In this way the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2 [see T_OPEN(NS_LIB) or T_GETINFO(NS_LIB)], **t_alloc** will be unable to determine the size of the buffer to allocate and will fail, setting **t_errno** to **TSYSERR** and **errno** to **EINVAL**. For any field not specified in **fields**, **buf** will be set to NULL and **maxlen** will be set to zero.

Use of **t_alloc** to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface functions.

ERRORS

On failure, **t_errno** is 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.

RETURN VALUE

On successful completion, **t_alloc** returns a pointer to the newly allocated structure. On failure, NULL is returned.

SEE ALSO

T_FREE(NS_LIB), T_GETINFO(NS_LIB), T_OPEN(NS_LIB).

NAME

t_bind - bind an address to a transport endpoint

SYNOPSIS

#include <tiuser.h>

int t_bind(fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;

DESCRIPTION

This function associates a protocol address with the transport endpoint specified by **fd** and activates that transport endpoint. In connection mode, the transport provider may begin accepting or requesting connections on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The **req** and **ret** arguments point to a **t_bind** structure containing the following members:

```
struct netbuf addr;
unsigned qlen;
```

The **addr** field of the **t_bind** structure specifies a protocol address and the **qlen** field is used to indicate the maximum number of outstanding connect indications.

Req is used to request that an address, represented by the **netbuf** structure, be bound to the given transport endpoint. **Len** specifies the number of bytes in the address and **buf** points to the address buffer. **Maxlen** has no meaning for the **req** argument. On return, **ret** contains the address that the transport provider actually bound to the transport endpoint; this may be different from the address specified by the user in **req**. 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, **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, or if no address is specified in **req** (the **len** field of **addr** in **req** is zero) the transport provider will assign an appropriate address to be bound, and will return that address in the **addr** field of **ret**. The user can compare the addresses in **req** and **ret** to determine whether the transport provider bound the transport endpoint to a different address than that requested.

Req may be NULL if the user does not wish to specify an 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 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 zero 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.

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 **qlen** greater than zero. 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 **qlen** greater than zero, the transport provider will assign another address to be bound to that endpoint. 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 connect indications.

ERRORS

On failure, **t_errno** is 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 allowed for an incoming argument is 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.

RETURN VALUE

T_bind returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_ALLOC(NS_LIB), T_OPEN(NS_LIB), T_OPTMGMT(NS_LIB), T_UNBIND(NS_LIB).

NAME

t_close - close a transport endpoint

SYNOPSIS

#include <tiuser.h>

int t_close(fd)
int fd;

DESCRIPTION

The **t_close** function informs the transport provider that the user is finished with the transport endpoint specified by **fd**, and frees any local library resources associated with the endpoint. In addition, **t_close** closes the file associated with the transport endpoint.

T_close should be called from the T_UNBND state [see T_GETSTATE(NS_LIB)]. However, this function does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, CLOSE(BA_OS) will be issued for that file descriptor; the close will be abortive if no other process has that file open, and will break any transport connection that may be associated with that endpoint.

ERRORS

On failure, **t_errno** is set to the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

RETURN VALUE

T_close returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_GETSTATE(NS_LIB), T_OPEN(NS_LIB), T_UNBIND(NS_LIB).

NAME

t_connect - establish a connection with another transport user

SYNOPSIS

#include <tiuser.h>

int t_connect(fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;

DESCRIPTION

This function enables a transport user to request a connection to the specified destination transport user. **Fd** identifies the local transport endpoint where communication will be established, while **sndcall** and **rcvcall** point to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

Sndcall specifies information needed by the transport provider to establish a connection and **rcvcall** specifies information that is associated with the newly established connection.

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 zero. 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(NS_LIB) or T_GETINFO(NS_LIB). If the **len** of **udata** is zero 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**.

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 (i.e., return value of zero) indicates that the requested connection has been established. However, if **O_NDELAY** is set [via T_OPEN(NS_LIB) or FCNTL(BA_OS)], **t_connect** executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return -1 with **t_errno** set to **TNODATA** to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user. The T_RCVCONNECT(NS_LIB) function is used in conjunction with **t_connect** to determine the status of the requested connection.

ERRORS

On failure, **t_errno** is 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_NDELAY** was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.
[TBADADDR]	The specified protocol address was in an incorrect format or contained illegal information.
[TBADOPT]	The specified protocol options were in an incorrect format or contained illegal information.

[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider.
[TACCES]	The user does not have permission to use the specified address or options.
[TBUFOVFLW]	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the connect indication information to be returned in **rcvcall** is discarded.
[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.

RETURN VALUE

T_connect returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_ACCEPT(NS_LIB), T_ALLOC(NS_LIB), T_GETINFO(NS_LIB), T_LISTEN(NS_LIB), T_OPEN(NS_LIB), T_OPTMGMT(NS_LIB), T_RCVCONNECT(NS_LIB).

NAME

t_error - produce error message

SYNOPSIS

```
#include <tiuser.h>

void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

DESCRIPTION

The **t_error** function produces a message on the standard error output which describes the last error encountered during a call to a transport function. The argument string **errmsg** is a user-supplied error message that gives context to the error.

T_error prints the user-supplied error message followed by a colon and a standard error message for the current error defined in **t_errno**. If **t_errno** is TSYSERR, **t_error** will also print a standard error message for the current value contained in **errno** [see INTRO(BA_OS)].

To simplify variant formatting of messages, the array of message strings **t_errlist** is provided; **t_errno** can be used as an index in this table to get the message string without the newline. **T_nerr** is the largest message number provided for in the **t_errlist** table.

T_errno is only set when an error occurs and is not cleared on successful calls.

EXAMPLE

If a T_CONNECT(NS_LIB) function fails on transport endpoint **fd2** because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd2");
```

The diagnostic message to be printed would look like:

```
t_connect failed on fd2:    Incorrect transport
                            address format
```

where **"Incorrect transport address format"** identifies the specific error that occurred, and **"t_connect failed on fd2"** tells the user which function failed on which transport endpoint.

NAME

t_free - free a library structure

SYNOPSIS

#include <tiuser.h>

int t_free(ptr, struct_type)
char *ptr;
int struct_type;

DESCRIPTION

The **t_free** function frees memory previously allocated by T_ALLOC(NS_LIB). This function will free memory for the specified structure, and will also free memory for buffers referenced by the structure.

Ptr points to one of the seven structure types described for T_ALLOC(NS_LIB), and **struct_type** identifies the type of that structure which must be one of the following:

`T_BIND`	`struct t_bind`
`T_CALL`	`struct t_call`
`T_OPTMGMT`	`struct t_optmgmt`
`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(NS_LIB).

ERRORS

On failure, **t_errno** is set to the following:

[TSYSERR] A system error has occurred during execution of this function.

RETURN VALUE

T_free returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_ALLOC(NS_LIB).

NAME

t_getinfo - get protocol-specific service information

SYNOPSIS

#include <tiuser.h>

int t_getinfo(fd, info)
int fd;
struct t_info *info;

DESCRIPTION

This function returns the current characteristics of the underlying transport protocol associated with file descriptor **fd**. The **info** structure is used to return the same information returned by T_OPEN(NS_LIB). This function enables a transport user to access this information during any phase of communication.

This argument points to a **t_info** structure which contains the following members:

```
long addr;          /* max size of the transport protocol */
                    /* address */
long options;       /* max number of bytes of */
                    /* protocol-specific options */
long tsdu;          /* max size of a transport service data */
                    /* unit (TSDU) */
long etsdu;         /* max size of an expedited transport */
                    /* service data unit (ETSDU) */
long connect;       /* max amount of data allowed on */
                    /* connection establishment functions */
long discon;        /* max amount of data allowed on */
                    /* t_snddis and t_rcvdis functions */
long servtype;      /* service type supported by the */
                    /* transport provider */
```

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; 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 zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero 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 zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero 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 zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; 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 zero specifies the maximum amount of data that may be associated with the T_SNDDIS(NS_LIB) and T_RCVDIS(NS_LIB) functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release 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, as described below.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the T_ALLOC(NS_LIB) function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of option negotiation, and T_GETINFO(NS_LIB) enables a user to retrieve the current characteristics of the underlying transport protocol.

The **servtype** field of **info** specifies one of the following values on return:

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 connectionlecs-mode service. For this service type, T_OPEN(NS_LIB) will return -2 for **etsdu**, **connect**, and **discon**.

ERRORS

On failure, **t_errno** is 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.

RETURN VALUE

T_getinfo returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_OPEN(NS_LIB).

NAME

t_getstate - get the current state

SYNOPSIS

#include <tiuser.h>

int t_getstate(fd)
int fd;

DESCRIPTION

The **t_getstate** function returns the current state of the provider associated with the transport endpoint specified by **fd**.

ERRORS

On failure, **t_errno** is 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 or **t_getstate** was called after an **exec**, **t_sync** sequence.
[TSYSERR]	A system error has occurred during execution of this function.

RETURN VALUE

T_getstate returns the current state on successful completion and -1 on failure and **t_errno** is set to indicate the error. The current state is one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication)
T_INREL	incoming orderly release (waiting to send an orderly release request)

If the provider is undergoing a state transition when **t_getstate** is called, the function will fail.

SEE ALSO

T_OPEN(NS_LIB).

NAME

t_listen - listen for a connect request

SYNOPSIS

#include <tiuser.h>

int t_listen(fd, call)
int fd;
struct t_call *call;

DESCRIPTION

This function listens for a connect request from a calling transport user. **Fd** identifies the local transport endpoint where connect indications arrive, and on return, **call** contains information describing the connect indication. **Call** points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

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 the **t_listen** to indicate the maximum size of the buffer for each.

By default, **t_listen** executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if **O_NDELAY** is set [via T_OPEN(NS_LIB) or FCNTL(BA_OS)], **t_listen** executes asynchronously, reducing to a poll for existing connect indications. If none are available, it returns -1 and sets **t_errno** to **TNODATA**.

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TBUFOVFLW] The number of bytes allocated for an incoming argument is 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_NDELAY** 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.

CAVEATS

If a user issues **t_listen** in synchronous mode on a transport endpoint that was not bound for listening [i.e., **qlen** was zero on T_BIND(NS_LIB)], the call will wait forever because no connect indications will arrive on that endpoint.

RETURN VALUE

T_listen returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_ACCEPT(NS_LIB), T_ALLOC(NS_LIB), T_BIND(NS_LIB),
T_CONNECT(NS_LIB), T_OPEN(NS_LIB), T_RCVCONNECT(NS_LIB).

NAME

t_look - look at the current event on a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_look(fd)
int fd;
```

DESCRIPTION

This function returns the current event on the transport endpoint specified by **fd**. This function enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. 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.

ERRORS

On failure, **t_errno** is 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.

RETURN VALUE

Upon success, **t_look** returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

T_LISTEN	connection indication received
T_CONNECT	connect confirmation received
T_DATA	normal data received
T_EXDATA	expedited data received
T_DISCONNECT	disconnect received
T_ERROR	fatal error indication
T_UDERR	datagram error indication
T_ORDREL	orderly release indication

On failure, -1 is returned, and **t_errno** is set to indicate the error.

SEE ALSO

T_OPEN(NS_LIB).

NAME

t_open - establish a transport endpoint

SYNOPSIS

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

int t_open(path, oflag, info)
char *path;
int oflag;
struct t_info *info;
```

DESCRIPTION

T_open must be called as the first step in the initialization of a transport endpoint. This function establishes a transport endpoint by opening a UNIX system file that identifies a particular transport provider (i.e., transport protocol) and returning a file descriptor that identifies that endpoint. For example, opening the file **/dev/iso_cots** identifies an OSI connection-oriented transport layer protocol as the transport provider.

Path points to the path name of the file to open, and **oflag** identifies any open flags [as in OPEN(BA_OS)]. **Oflag** may be constructed from **O_NDELAY** or-ed with either **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These flags are defined by the header file **<fcntl.h>**. **T_open** returns a file descriptor that will be used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the **t_info** structure. This argument points to a **t_info** which contains the following members:

```
long addr;        /* max size of the transport protocol */
                  /* address */
long options;     /* max number of bytes of */
                  /* protocol-specific options */
long tsdu;        /* max size of a transport service data */
                  /* unit (TSDU) */
long etsdu;       /* max size of an expedited transport */
                  /* service data unit (ETSDU) */
long connect;     /* max amount of data allowed on */
                  /* connection establishment functions */
long discon;      /* max amount of data allowed on */
                  /* t_snddis and t_rcvdis functions */
long servtype;    /* service type supported by the */
                  /* transport provider */
```

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; 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 zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero 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 zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero 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 zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; 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 zero specifies the maximum amount of data that may be associated with the T_SNDDIS(NS_LIB) and T_RCVDIS(NS_LIB) functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release 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, as described below.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the T_ALLOC(NS_LIB) function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

The **servtype** field of **info** specifies one of the following values on return:

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-mode service. For this service type, **t_open** will return -2 for **etsdu**, **connect**, and **discon**.

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**.

ERRORS

On failure, **t_errno** is set to the following:

[TSYSERR] A system error has occurred during execution of this function.

RETURN VALUE

T_open returns a valid file descriptor on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

OPEN(BA_OS).

NAME

t_optmgmt - manage options for a transport endpoint

SYNOPSIS

#include <tiuser.h>

int t_optmgmt(fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;

DESCRIPTION

The **t_optmgmt** function enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider. **Fd** identifies a bound transport endpoint.

The **req** and **ret** arguments point to a **t_optmgmt** structure containing the following members:

```
struct netbuf opt;
long    flags;
```

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 in T_BIND(NS_LIB). **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** must 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.

If issued as part of the connectionless-mode 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.

ERRORS

On failure, **t_errno** is 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 allowed for an incoming argument is 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.

RETURN VALUE

T_optmgmt returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_ALLOC(NS_LIB), T_GETINFO(NS_LIB), T_OPEN(NS_LIB).

NAME

t_rcv - receive data or expedited data sent over a connection

SYNOPSIS

int t_rcv(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;

DESCRIPTION

This function receives either normal or expedited data. **Fd** identifies the local transport endpoint through which data will arrive, **buf** points to a receive buffer where user data will be placed, and **nbytes** specifies the size of the receive buffer. **Flags** may be set on return from **t_rcv** and specifies optional flags as described below.

By default, **t_rcv** operates in synchronous mode and will wait for data to arrive if none is currently available. However, if **O_NDELAY** is set [via T_OPEN(NS_LIB) or FCNTL(BA_OS)], **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 immediately 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(NS_LIB) or T_GETINFO(NS_LIB), the **T_MORE** flag is not meaningful and should be ignored.

On return, the data returned is expedited data if **T_EXPEDITED** is set in **flags**. 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.

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNODATA]	**O_NDELAY** 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.

RETURN VALUE

On successful completion, **t_rcv** returns the number of bytes received; it returns -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_OPEN(NS_LIB), T_SND(NS_LIB).

NAME

t_rcvconnect - receive the confirmation from a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_rcvconnect(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

This function enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with T_CONNECT(NS_LIB) to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

Fd identifies the local transport endpoint where communication will be established, and **call** contains information associated with the newly established connection. **Call** points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

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**. By default, **t_rcvconnect** executes in synchronous mode and waits for the connection to be established before returning. On return, the **addr**, **opt**, and **udata** fields reflect values associated with the connection.

If **O_NDELAY** is set [via T_OPEN(NS_LIB) or FCNTL(BA_OS)], **t_rcvconnect** executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, **t_rcvconnect** fails and returns immediately without waiting for the connection to be established. (See **TNODATA** below.) **T_rcvconnect** must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in **call**.

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument and 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_NDELAY** was set, but a connect confirmation has not yet arrived.
[TLOOK]	An asynchronous event has occurred on this transport connection 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.

RETURN VALUE

T_rcvconnect returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_ACCEPT(NS_LIB), T_ALLOC(NS_LIB), T_BIND(NS_LIB), T_CONNECT(NS_LIB), T_LISTEN(NS_LIB), T_OPEN(NS_LIB).

NAME

t_rcvdis - retrieve information from disconnect

SYNOPSIS

#include <tiuser.h>

t_rcvdis(fd, discon)
int fd;
struct t_discon *discon;

DESCRIPTION

This function is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. **Fd** identifies the local transport endpoint where the connection existed, and **discon** points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

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(NS_LIB) 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(NS_LIB)] and **discon** is NULL, the user will be unable to identify with which connect indication the disconnect is associated.

ERRORS

On failure, **t_errno** is 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 allocated for incoming data is not sufficient to store the data. 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.

RETURN VALUE

T_rcvdis returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_ALLOC(NS_LIB), T_CONNECT(NS_LIB), T_LISTEN(NS_LIB), T_OPEN(NS_LIB), T_SNDDIS(NS_LIB).

NAME

t_rcvrel - acknowledge receipt of an orderly release indication

SYNOPSIS

#include <tiuser.h>

t_rcvrel(fd)
int fd;

DESCRIPTION

This function is used to acknowledge receipt of an orderly release indication. **Fd** identifies the local transport endpoint where the connection exists. After receipt of this indication, the user may 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(NS_LIB) 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(NS_LIB) or T_GETINFO(NS_LIB).

ERRORS

On failure, **t_errno** is 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 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.

RETURN VALUE

T_rcvrel returns 0 on success and -1 on failure with **t_errno** set to indicate the error.

SEE ALSO

T_OPEN(NS_LIB), T_SNDREL(NS_LIB).

NAME

t_rcvudata - receive a data unit

SYNOPSIS

```
#include <tiuser.h>

int t_rcvudata(fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

DESCRIPTION

This function is used in connectionless mode to receive a data unit from another transport user. **Fd** identifies the local transport endpoint through which data will be received, **unitdata** holds information associated with the received data unit, and **flags** is set on return to indicate that the complete data unit was not received. **Unitdata** points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

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.

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_NDELAY** is set [via T_OPEN(NS_LIB) or FCNTL(BA_OS)], **t_rcvudata** will execute in asynchronous mode and will fail if no data units are available.

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 zero for the length of the address and options until the full data unit has been received.

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNODATA]	**O_NDELAY** was set, but no data units are currently available from the transport provider.
[TBUFOVFLW]	The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data information to be returned in **unitdata** will be discarded.
[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.

RETURN VALUE

T_rcvudata returns 0 on successful completion and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_ALLOC(NS_LIB), T_RCVUDERR(NS_LIB), T_SNDUDATA(NS_LIB).

NAME

t_rcvuderr - receive a unit data error indication

SYNOPSIS

#include <tiuser.h>

int t_rcvuderr(fd, uderr)
int fd;
struct t_uderr *uderr;

DESCRIPTION

This function 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.

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TNOUDERR] No unit data error indication currently exists on the specified transport endpoint.

[TBUFOVFLW] 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]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

RETURN VALUE

T_rcvuderr returns 0 on successful completion and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_RCVUDATA(NS_LIB), T_SNDUDATA(NS_LIB).

NAME

t_snd - send data or expedited data over a connection

SYNOPSIS

```
#include <tiuser.h>

int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

DESCRIPTION

This function is used to send either normal or expedited data. **Fd** identifies the local transport endpoint over which data should be sent, **buf** points to the user data, **nbytes** specifies the number of bytes of user data to be sent, and **flags** specifies any optional flags described below.

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_NDELAY** is set [via T_OPEN(NS_LIB) or FCNTL(BA_OS)], **t_snd** will execute in asynchronous mode, and will fail immediately if there are flow control restrictions.

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_NDELAY** is set, it is possible that only part of the data will actually 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 that is less than the value of **nbytes**. If **nbytes** is zero, no data will be passed to the provider, and **t_snd** will return zero.

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 as described above, this indicates 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(NS_LIB) or T_GETINFO(NS_LIB), 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 in the TSDU or ETSDU fields of the **info** argument of T_OPEN(NS_LIB) or T_GETINFO(NS_LIB). Failure to comply will result in protocol error **EPROTO**. (See **TSYSERR** below.)

If **t_snd** is issued from the T_IDLE state, the provider may silently discard the data. If **t_snd** is issued from any state other than T_DATAXFER, T_INREL, or T_IDLE, the provider will generate an **EPROTO** error.

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TFLOW] **O_NDELAY** 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_snd** to fail until a subsequent access of the transport endpoint.

RETURN VALUE

On successful completion, **t_snd** returns the number of bytes accepted by the transport provider; it returns -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_OPEN(NS_LIB), T_RCV(NS_LIB).

NAME

t_snddis - send user-initiated disconnect request

SYNOPSIS

#include <tiuser.h>

int t_snddis(fd, call)
int fd;
struct t_call *call;

DESCRIPTION

This function is used to initiate an abortive release on an already established connection or to reject a connect request. **Fd** identifies the local transport endpoint of the connection, and **call** specifies information associated with the abortive release. **Call** points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The values in **call** have different semantics, depending on the context of the call to **t_snddis**. When rejecting a connect request, **call** must be non-NULL and contain a valid value of **sequence** to uniquely identify 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 wish 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(NS_LIB) or T_GETINFO(NS_LIB). If the **len** field of **udata** is zero, no data will be sent to the remote user.

ERRORS

On failure, **t_errno** is 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 allowed by the transport provider. 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 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.

RETURN VALUE

T_snddis returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_CONNECT(NS_LIB), T_GETINFO(NS_LIB), T_LISTEN(NS_LIB), T_OPEN(NS_LIB).

NAME

t_sndrel - initiate an orderly release

SYNOPSIS

```
#include <tiuser.h>

int t_sndrel(fd)
int fd;
```

DESCRIPTION

This function is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send. **Fd** identifies the local transport endpoint where the connection exists. 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 been received.

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(NS_LIB) or T_GETINFO(NS_LIB).

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TFLOW]	**O_NDELAY** 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.

RETURN VALUE

T_sndrel returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_OPEN(NS_LIB), T_RCVREL(NS_LIB).

NAME

t_sndudata - send a data unit

SYNOPSIS

#include <tiuser.h>

int t_sndudata(fd, unitdata)
int fd;
struct t_unitdata *unitdata;

DESCRIPTION

This function is used in connectionless mode to send a data unit to another transport user. **Fd** identifies the local transport endpoint through which data will be sent, and **unitdata** points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

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 zero. In this case, the provider may use default options.

If the **len** field of **udata** is zero, no data unit will be passed to the transport provider; **t_sndudata** will not send zero-length data units.

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_NDELAY** is set [via T_OPEN(NS_LIB) or FCNTL(BA_OS)], **t_sndudata** will execute in asynchronous mode and will fail under such conditions.

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(NS_LIB) or T_GETINFO(NS_LIB), the provider will generate an **EPROTO** protocol error. (See **TSYSERR** below.) If **t_sndudata** is issued before the destination user has activated its transport endpoint [see T_BIND(NS_LIB)], the data unit may be discarded.

ERRORS

On failure, **t_errno** is set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TFLOW]	**O_NDELAY** 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 a subsequent access of the transport endpoint.

RETURN VALUE

T_sndudata returns 0 on successful completion and -1 on failure; **t_errno** is set to indicate the error.

SEE ALSO

T_ALLOC(NS_LIB), T_RCVUDATA(NS_LIB), T_RCVUDERR(NS_LIB).

NAME

t_sync - synchronize transport library

SYNOPSIS

#include <tiuser.h>

int t_sync(fd)
int fd;

DESCRIPTION

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 [obtained via OPEN(BA_OS), DUP(BA_OS), or as a result of a FORK(BA_OS) and EXEC(BA_OS)] to an initialized transport endpoint, assuming that 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 structure with the relevant provider information.

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, they should coordinate their activities so as not to violate the state of the provider. **T_sync** returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the provider's state *after* a **t_sync** is issued.

If the provider is undergoing a state transition when **t_sync** is called, the function will fail.

ERRORS

On failure, **t_errno** is 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.

RETURN VALUE

T_sync returns the state of the transport provider on successful completion and -1 on failure; **t_errno** is set to indicate the error. The state returned is one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication)
T_INREL	incoming orderly release (waiting for an orderly release request)

SEE ALSO

DUP(BA_OS), EXEC(BA_OS), FORK(BA_OS), OPEN(BA_OS).

NAME

t_unbind - disable a transport endpoint

SYNOPSIS

#include <tiuser.h>

int t_unbind(fd)
int fd;

DESCRIPTION

The **t_unbind** function disables the transport endpoint specified by **fd** which was previously bound by T_BIND(NS_LIB). On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

ERRORS

On failure, **t_errno** is 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.
[TLOOK]	An asynchronous event has occurred on this transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.

RETURN VALUE

T_unbind returns 0 on success and -1 on failure, and **t_errno** is set to indicate the error.

SEE ALSO

T_BIND(NS_LIB).

Chapter 14
Streams I/O Interfaces

14.1 INTRODUCTION

The STREAMS I/O INTERFACES section of the NETWORK SERVICES EXTENSION describes the interfaces that enable a user to directly access protocol modules that are implemented in the kernel using the STREAMS framework. STREAMS provides a uniform mechanism for implementing network services in the kernel by defining standard interfaces for device drivers and protocol modules.

This extension is dependent on the Base System.

14.2 DESCRIPTION

OPERATING SYSTEM SERVICE ROUTINES

getmsg	**poll**	**putmsg**

HEADER FILES

poll.h	**stropts.h**

ERROR CONDITIONS

EBADMSG	Trying to read unreadable message
ENOSR	Out of stream resources
ENOSTR	Device not a stream
EPROTO	Protocol error occurred
ETIME	Timer expired

14.3 DEFINITIONS

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.

Module and 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.

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.

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.

STREAMS Messages

STREAMS I/O is based on messages. Message types are classified according to their queueing priority and may be non-priority messages or priority messages. Non-priority messages are always placed at the end of the queue following all other messages in the queue. Priority messages are always placed at the head of a queue but after any other priority messages already in the queue. Priority messages are used to send control and data information outside the normal flow control constraints. A user may access STREAMS messages that contain a data part, control part, or both. The data part 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 are not accessible to users.

strbuf Structure

The **strbuf** structure is used to contain data or control information and is used by the **getmsg**, **putmsg**, and **ioctl** operating system service routines. This structure is defined by the header file **stropts.h** and includes the following members:

```
int maxlen;    /* maximum buffer length */
int len;       /* length of data */
char *buf;     /* ptr to data buffer */
```

14.4 EFFECTS ON THE BASE SYSTEM

Components in the Base System may return a new value for **errno** as listed below. An application that checks the value of **errno** must include the header file <**errno.h**>.

The following symbolic names define additional error return conditions:

Name	Description
EBADMSG	Trying to read unreadable message
ENOSR	Out of stream resources
ENOSTR	Device not a stream
EPROTO	Protocol error
ETIME	Timer expired

These errors may be returned by the operating system service routines **open**, **close**, **read**, **write**, **ioctl**, **getmsg**, **putmsg**, and **poll** only when accessing STREAMS devices and as described in the detailed definitions of the components that follow the detailed overview.

A new signal has been defined by the header file <**signal.h**>. This signal is used to support asynchronous processing of events on STREAMS devices.

The following symbolic name defines the additional signal:

Name	Description
SIGPOLL	Signals STREAMS events

14.5 OVERVIEW

STREAMS is a general, flexible facility for development of UNIX system 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 that is also available in the UNIX system.

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). For these protocol suites, developers have traditionally faced problems arising from lack of relevant standard interfaces in the UNIX system. 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.

---

* Xerox is a registered trademark of Xerox Corporation.

- 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.

Implementing networking facilities and communication components under STREAMS allows efficient, open ended products.

"STREAMS" refers to the mechanism consisting of operating system service routines, kernel resources and kernel utility routines. A stream, as illustrated in Figure 14-1, is a full duplex processing and data transfer path in the kernel that is created through an application of the STREAMS mechanism.

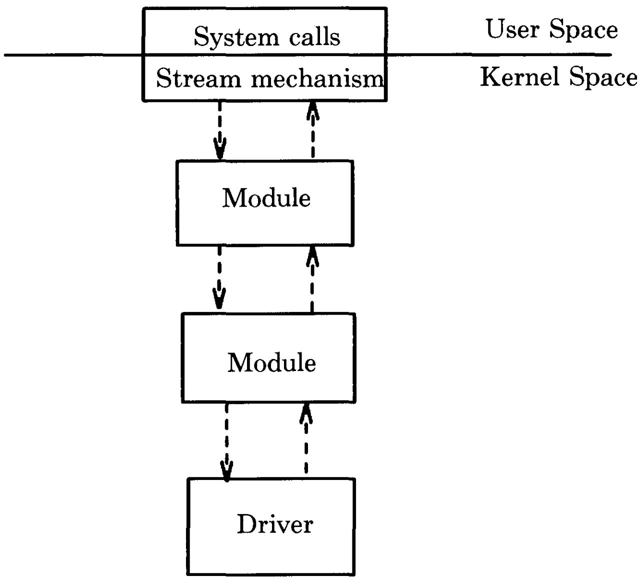

Figure 14-1: Basic Stream

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 and these are classified according to queueing priority. Non-priority messages are always placed at the end of the queue following all other messages in the queue. Priority messages are always placed at the head of a queue but after any other priority messages already in the queue. Priority messages are used to send control and data information outside the normal flow control constraints. However, to prevent congestion and resource waste due to lack of flow control with this message type, only one 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**, **getmsg**, **read**, and **write** routines. Messages containing a control portion with or without a data portion are accessible via calls to **putmsg** and **getmsg**.

The interface between a user process and STREAMS is compatible with the existing character I/O facilities, and both are available in the UNIX system.

14.6 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 new routines **putmsg**, **getmsg**, and **poll**.

14.6.1 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 14-2, 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.

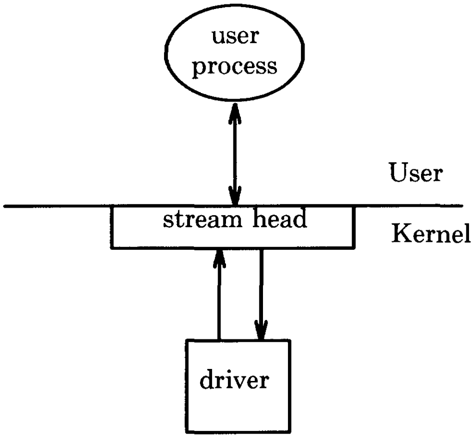

Figure 14-2: Setting Up a Stream

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 Figure 14-3.

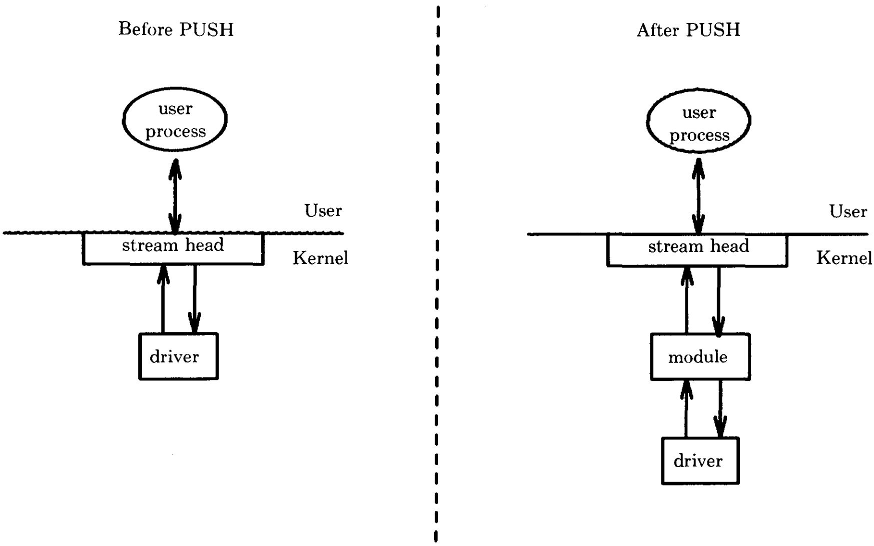

Figure 14-3: Before and After a Module is Pushed

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 path name, while modules are "pushed" onto the stream by module name. Device path names are ordinary UNIX system file names, but pushable modules' names are internal to the system and are not opened or closed.

14.6.2 Sending and Receiving STREAMS Messages

In order to send and receive STREAMS messages that contain control information, the new routines **getmsg** and **putmsg** must be used. These differ from **read** and **write** in that the traditional routines can access non-priority STREAMS messages containing only data, while **getmsg** and **putmsg** can access priority and non-priority 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.

14.6.3 Polling STREAMS

The **poll** routine provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open STREAMS. It 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 event specific to the **poll** operating system service routine.

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**.

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.

14.7 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.

14.7.1 Setting Up a Multiplexer

A multiplexing driver is a pseudo-device, and is treated like any other software driver. It owns a node in the UNIX system 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 14-4 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 UNIX system device. An **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 14-4).

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 14-4), 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 argument **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 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 14-4.) 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 14-4 shows two drivers and a multiplexing driver before and after the two drivers have been linked below the multiplexer.

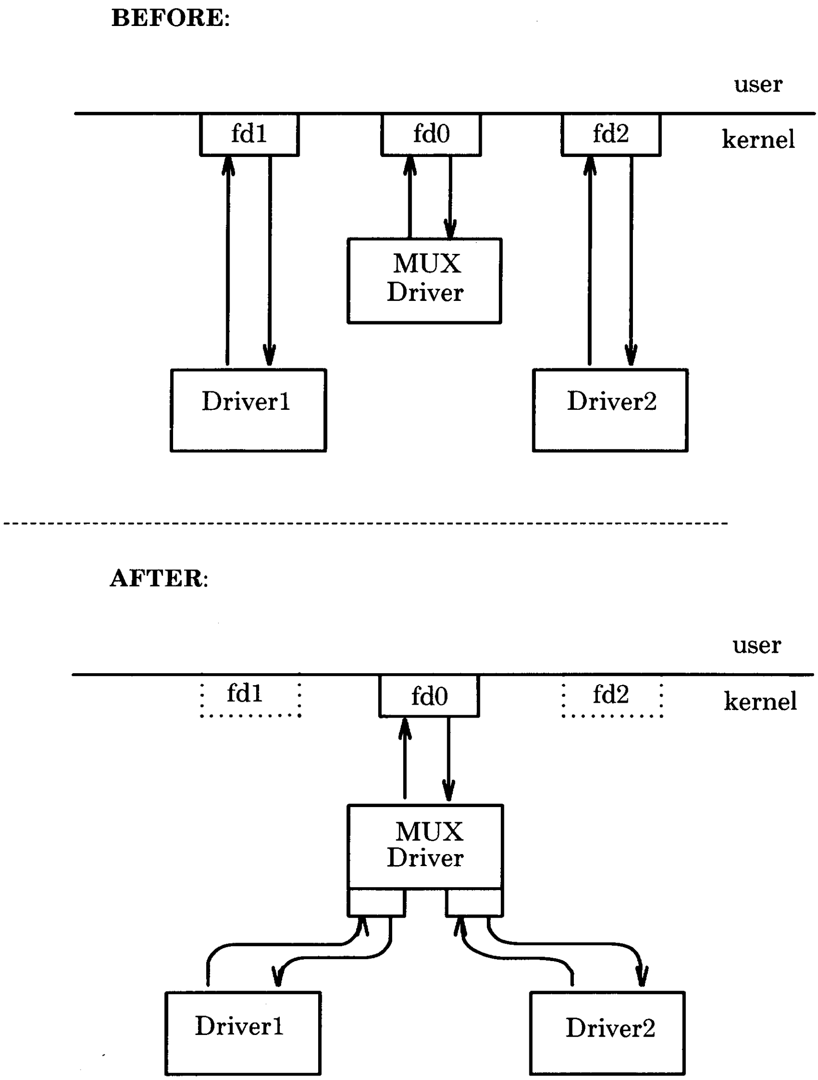

Figure 14-4: A Multiplexing Configuration Before and After 2 **I_LINK ioctls**

Other device drivers are opened and linked below the multiplexing driver in the same way, as in the example shown in Figure 14-4:

```
/* 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 **ioctl I_LINK** 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 14-4) can be closed without affecting the way the multiplexer works; these closes will not cause the drivers to be unlinked from the multiplexer. Closing these file descriptors is necessary sometimes when building large multiplexers, so that many devices can be linked together without exceeding the UNIX system limit on the number of simultaneously-open files per process. If these file descriptors (**fd1** and **fd2** in Figure 14-4) are not closed, the multiplexer will work as expected, but all subsequent **read**, **write**, **poll**, **putmsg**, and **getmsg** UNIX operating system service routines issued to **fd1** and **fd2** will fail.

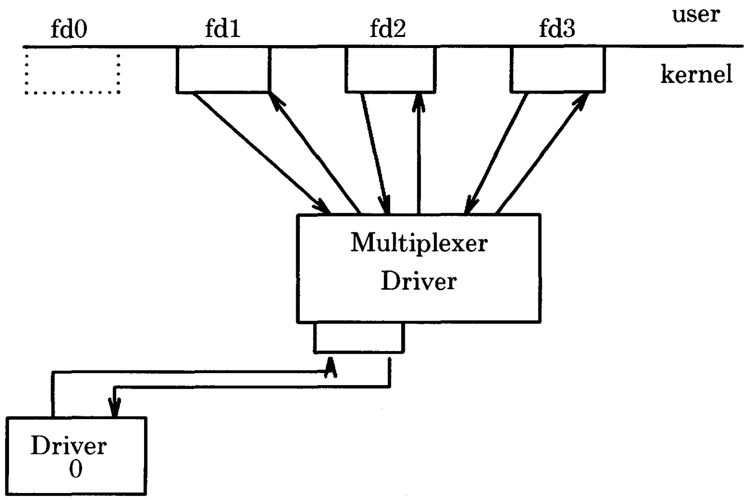

Figure 14-5: Three STREAMS Converging on One Device Driver

Building a multiplexer that connects several STREAMS to a single driver, as in Figure 14-5, is similar, except that only one driver is linked below the multiplexer. Additional STREAMS above the multiplexer would be established by issuing repeated **open** operating system service routines to the multiplexer on "related" minor devices. Again, the way the multiplexer handles these repeated **opens** 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 14-4 and 14-5 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 stream to the other.

14.7.2 Dismantling a Multiplexer

Multiplexing configurations are taken apart using the **ioctl I_UNLINK** command. Each of the bottom drivers linked below the multiplexing driver (e.g., Driver1 and Driver2 in Figure 14-4) 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 -1 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 driver; all these bottom STREAMS are then unlinked.

14.7.3 Multiplexed Data Routing

Processes use the normal UNIX system **read**, **write**, **getmsg**, and **putmsg** 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 driver has access to this value, and the **I_LINK ioctl** command returns this value to the user. The driver can therefore specify that the "mux id" value accompany the data routed through it.

NAME

streams - STREAMS interface

DESCRIPTION

STREAMS provides a uniform mechanism for implementing networking services and other I/O in the kernel. The STREAMS interface provides direct access to protocol modules that are implemented in the kernel. A user process accesses STREAMS using the standard operating system service routines described below as well as the new routines PUTMSG(NS_OS), GETMSG(NS_OS), and POLL(NS_OS). 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.

STREAMS I/O is based on messages. Messages flow in both directions in a stream. A given module may not understand and process every message in the stream, but every module in the stream handles every message. Each module accepts messages from one of its neighbor modules in the stream, and passes them to the other neighbor. A line discipline module may transform the data. Data flow through the intermediate modules is symmetrical, with all modules handling, and optionally processing, all messages.

The interface between the stream and the rest of the operating system is provided by a set of routines at the stream head (upstream) end of the stream. User-process WRITE(BA_OS), PUTMSG(NS_OS), and IOCTL(BA_OS) calls become messages that are sent down the stream, and the READ(BA_OS) and GETMSG(NS_OS) calls accept data from the stream and pass it to a user process. Data intended for the device at the downstream end of the stream is packaged into messages and sent downstream, while data and signals from the device are composed into messages by the device driver and sent upstream to the stream head.

When a device is opened, the system creates a stream that contains two modules: the stream head module and the stream end (driver) module. Other modules are added to the stream using the IOCTL(BA_OS) routine. New modules are "pushed" onto the stream one at a time in last-in, first-out (LIFO) style, as though the stream was a push-down stack.

There are many message types used by STREAMS modules and these are classified according to queueing priority. Non-priority messages are always placed at the end of the queue following all other messages in the queue. Priority messages are always placed at the head of a queue but after any other priority messages already in the queue. Priority messages are used to send control and data information outside the normal flow control constraints. A user may access STREAMS messages that contain a data part, control part, or both. The data part 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 part are accessible via **putmsg**, **getmsg**, **read**, and **write** routines. Messages containing a control part with or without a data part are accessible via calls to **putmsg** and **getmsg**.

Accessing STREAMS Devices

A user process accesses STREAMS devices using the standard routines OPEN(BA_OS), CLOSE(BA_OS), READ(BA_OS), WRITE(BA_OS), and IOCTL(BA_OS) routines as well as the new routines PUTMSG(NS_OS), GETMSG(NS_OS), and POLL(NS_OS). Refer to the detailed component definitions for **open, close, read, write**, and **ioctl** for general properties and errors.

Open calls [see OPEN(BA_OS)] have the format

```
int open (path, oflag)
char *path;
int oflag;
```

When opening a STREAMS file, **oflag** may be constructed from **O_NDELAY** or-ed with either **O_RDONLY, O_WRONLY**, or **O_RDWR**. These values are defined by <**fcntl.h**> so the line

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

must be included in the user program. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of **O_NDELAY** affects the operation of STREAMS drivers and certain system calls [see READ(BA_OS), GETMSG(NS_OS), PUTMSG(NS_OS), and WRITE(BA_OS)]. For drivers, the implementation of **O_NDELAY** is device-specific. Each STREAMS device driver may treat this option differently. Certain flag values can be set following **open** as described in FCNTL(BA_OS). On success, **open** returns a file descriptor that corresponds to the opened stream. On failure, **open** returns -1 and sets **errno** to one of the following:

[EINTR] A signal was caught during the **open**.

[ENXIO] A module or driver open routine failed.

[ENOSR] Unable to allocate a stream.

[EIO] A hangup or error occurred during the **open**.

Close [see CLOSE(BA_OS)] is used to close a device and calls have the format

```
int close (fildes)
int fildes;
```

If a STREAMS file is closed and the calling process had previously registered to receive a **SIGPOLL** signal [see SIGNAL(BA_OS) and SIGSET(BA_OS)] for events associated with that file, the calling process will be unregistered for events associated with the file. The last **close** for a **stream** causes the **stream** associated with **fildes** to be dismantled. If **O_NDELAY** is not set and there have been no signals posted for the **stream**, **close** waits up to 15 seconds (for each module and driver) for any output to drain before dismantling the **stream**. If the **O_NDELAY** flag is set or if there are any pending signals, **close** does not wait for output to drain, and dismantles the **stream** immediately. **Close** returns 0 on success. On failure, **close** returns -1 and sets **errno** to one of the following values:

[EBADF] **Fildes** is not a valid open file descriptor.

[EINTR] A signal was caught during the **close**.

The **read** routine [see READ(BA_OS)] attempts to read **nbyte** bytes of data from the file associated with **fildes** into the buffer pointed to by **buf**. **Read** calls have the format

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

Read can operate in three different modes: "byte-stream" mode, "message-nondiscard" mode, and "message-discard" mode. The default read mode is byte-stream mode. This can be changed using the **I_SRDOPT ioctl** request, and can be tested with the **I_GRDOPT ioctl**. In byte-stream mode, **read** will retrieve data from the **stream** until it has retrieved **nbyte** bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries. In message-nondiscard mode, read retrieves data until it has read **nbyte** bytes, or until it reaches a message boundary. If the **read** 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

getmsg) 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** returns is discarded, and is not available for a subsequent **read** (or **getmsg**) call.

When attempting to read a file associated with a **stream** that has no data currently available:

> If **O_NDELAY** is set, the read will return a -1 and set **errno** to **EAGAIN**.
>
> If **O_NDELAY** is clear, the read will block until data becomes available.

The **read** call's 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**. 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** from a STREAMS file can only process messages with data and without control information. The **read** will fail if a message containing control information is encountered at the **stream head**.

Read returns the number of bytes read when it succeeds. On failure, **read** returns -1 and sets **errno** to one of the following:

[EBADF]	**Fildes** is not a valid open file descriptor.
[EFAULT]	**Buf** points outside the allocated address space.
[EBADMSG]	Message waiting to be read is not a data message.
[EAGAIN]	No message waiting to be read, and **O_NDELAY** flag set.
[EINVAL]	Attempted to read from a **stream** linked to a multiplexer.
[EINTR]	A signal was caught during the **read**.

Read will also fail 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** will continue to operate normally until the stream head read queue is empty. Thereafter, it will return 0.

The **write** routine [see WRITE(BA_OS)] attempts to write **nbyte** bytes from the buffer **buf** to the device associated with the file descriptor **fildes.** **Write** calls have the format

```
int write(fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

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, these values cannot be set or tested from user level. If **nbyte** falls within the packet size range, **nbyte** bytes will be written. If **nbyte** does not fall within the range and the minimum packet size value is zero, **write** will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If **nbyte** does not fall within the range and the minimum value is non-zero, **write** will fail with **errno** set to ERANGE. Writing a zero-length buffer (**nbyte** is zero) sends zero bytes with zero returned.

If **O_NDELAY** is not set and the **stream** cannot accept data (the **stream** write queue is full due to internal flow control conditions), **write** will block until data can be accepted. If **O_NDELAY** is set and the **stream** cannot accept data, **write** will return -1 and set **errno** to **EAGAIN.** If **O_NDELAY** is set and part of the buffer has been written when a condition in which the **stream** cannot accept additional data occurs, **write** will terminate and return the number of bytes written. Upon successful completion, the number of bytes actually written is returned.

On failure, **write** returns -1 and sets **errno** to one of the following values:

[EBADF] **Fildes** is not a valid open file descriptor.

[EFAULT] **Buf** points outside the allocated address space.

[ERANGE] Attempt to write to a **stream** with **nbyte** outside specified minimum and maximum write range, and the minimum value is non-zero.

[EAGAIN] Attempt to write to a **stream** that cannot accept data with the **O_NDELAY** flag set.

[EINVAL] Attempt to read from a **stream** linked to a multiplexer.

[EINTR] A signal was caught during the **write**.

[ENXIO] A hangup occurred on the stream being written to.

Write can also 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.

Ioctl calls [see IOCTL(BA_OS)] are used to perform control functions with the device associated with the file descriptor **fildes**. The arguments **command** and **arg** are passed to the file designated by **fildes** and are interpreted by the **stream head**. Certain combinations of these arguments may be passed to a module or driver in the **stream**.

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.

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 multiplexer, 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. This causes subsequent system calls to fail with **errno** set to this value. **Ioctl** calls have the format

```
int ioctl(fildes, command, arg)
int fildes;
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 upon success and -1 upon 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 line

```
#include <stropts.h>
```

must be included in the user program.

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**. 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.
- **[EFAULT]** **Arg** points outside the allocated address space.
- **[ENXIO]** Open routine of new module failed.
- **[ENXIO]** Hangup received on **fildes**.

I_POP Removes the module just below the **stream head** of the **stream** pointed to by **fildes**. **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 character string pointed to by **arg**. The buffer pointed to by **arg** should be at least FMNAMESZ+1 bytes long where FMNAMESZ is defined by "**#include <sys/conf.h>**". On failure, **errno** is set to one of the following values:

- **[EFAULT]** **Arg** points outside the allocated address space.
- **[EINVAL]** No module present in **stream**.

I_FLUSH This request flushes all input and/or output queues, depending on the value of **arg**. Legal **arg** values are:

FLUSHR Flush read queues.

FLUSHW Flush write queues.

FLUSHRW Flush read and write queues.

On failure, **errno** is set to one of the following values:

[EINVAL] Invalid **arg** value.

[EAGAIN] Unable to allocate buffers for flush message.

[ENXIO] Hangup received on **fildes**.

I_SETSIG Informs the **stream head** that the user wishes the kernel to issue the **SIGPOLL** signal [see SIGNAL(BA_OS) and SIGSET(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 of the following constants:

S_INPUT A non-priority message has arrived on a **stream head** read queue, and no other messages existed on that queue before this message was placed there. This is set even if the message is of zero length.

S_HIPRI A priority message is present on a **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_MSG A STREAMS signal message that contains the **SIGPOLL** signal has reached the front of the **stream head** read queue.

A user process may choose to handle asynchronously only priority messages by setting the **arg** bitmask to the value **S_HIPRI.**

Processes that wish 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.

[EFAULT] **Arg** points outside the allocated address space.

I_FIND This request 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:

[EFAULT] **Arg** points outside the allocated address space.

[EINVAL] **Arg** does not contain a valid module name.

I_PEEK This request allows a user to retrieve the information in the first message on the **stream head** read queue without taking the message off the queue. **Arg** points to a **strpeek** structure which contains the following members:

```
struct strbuf                 ctlbuf;
struct strbuf                 databuf;
long                          flags;
```

where **strbuf** is a structure that contains the following members:

```
int maxlen;
int len;
char *buf;
```

The **maxlen** field in the **ctlbuf** and **databuf strbuf** structures [see GETMSG(NS_OS)] must be set to the number of bytes of control information and/or data information, respectively, to retrieve. If the user sets **flags** to **RS_HIPRI**, **I_PEEK** will only look for a priority 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, or if the **RS_HIPRI** flag was set in **flags** and a priority message was not present 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 0 or **RS_HIPRI**. On failure, **errno** is set to the following value:

[EFAULT] **Arg** points or the buffer area specified in **ctlbuf** or **databuf** is outside the allocated address space.

I_SRDOPT Sets the read mode using the value of the argument **arg**. Legal **arg** values are:

RNORM Byte-stream mode, the default.

RMSGD Message-discard mode.

RMSGN Message-nondiscard mode.

Read modes are described in READ(BA_OS). On failure, **errno** is set to the following value:

[EINVAL] **Arg** is not one of the above legal values.

I_GRDOPT Returns the current read mode setting in an **int** pointed to by the argument **arg**. Read modes are described in READ(BA_OS). On failure, **errno** is set to the following value:

[EFAULT] **Arg** points outside the allocated address space.

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. On failure, **errno** is set to the following value:

[EFAULT] **Arg** points outside the allocated address space.

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(NS_OS)] must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. **Fd** specifies the file descriptor of the other **stream** and **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 to the **fd stream**'s driver read queue 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. A non-priority message is created if **flags** is set to 0, and a priority message is created if **flags** is set to **RS_HIPRI**. For non-priority messages, **I_FDINSERT** will block if the **stream** write queue is full due to internal flow control conditions. For priority messages, **I_FDINSERT** does not block on this condition. For non-priority messages, **I_FDINSERT** does not block when the write queue is full and **O_NDELAY** 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 in the **stream**, regardless of priority or whether **O_NDELAY** 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_NDELAY** 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.

[EFAULT] **Arg** points, or the buffer area specified in **ctlbuf** or **databuf** is, outside the allocated address space.

[EINVAL] One of the following: **fd** 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**.

[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_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 system responds with either a positive or negative acknowledgment message, or until the request "times out" after some period of time. If the request times out, it fails with **errno** set to **ETIME**.

At most, one **I_STR** can be active on a **stream**. Further **I_STR** calls will block until the active **I_STR** completes at the **stream head**. The default timeout interval for these requests is 15 seconds. The **O_NDELAY** [see OPEN(BA_OS)] flag has no effect on this call.

To send requests downstream, **arg** must point to a **strioctl** structure which contains the following members:

```
int     ic_cmd;    /* downstream command */
int     ic_timout;/* ACK/NAK timeout */
int     ic_len;    /* length of data arg */
char    *ic_dp;    /* ptr to data arg */
```

Ic_cmd is the internal **ioctl** command intended for a downstream module or driver and **ic_timout** is the number of seconds (-1 = infinite, 0 = use default, >0 = as specified) an **I_STR** request will wait for acknowledgment before timing out. **Ic_len** is the number of bytes in the data argument, and **ic_dp** is a pointer to the data argument. The **ic_len** field has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the user (the buffer pointed to by **ic_dp** should be large enough to contain the maximum amount of data that any module or the driver in the **stream** can return).

The **stream head** will convert the information pointed to by the **strioctl** structure to an internal ioctl command message and send it downstream. On failure, **errno** is set to one of the following values:

[EAGAIN] Unable to allocate buffers for the **ioctl** message.

[EFAULT] **Arg** points or the buffer area specified by **ic_dp** and **ic_len** (separately for data sent and data returned) is, outside the allocated address space.

[EINVAL] **Ic_len** is less than 0, or **ic_len** is larger than the maximum configured size of the data part of a message, or **ic_timout** is less than -1.

[ENXIO] Hangup received on **fildes**.

[ETIME] A downstream **ioctl** timed out before acknowledgment was received.

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.

The following two 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 multiplexer ID number (an identifier used to disconnect the multiplexer, 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**.

[ETIME] Time out before acknowledgement message was received at **stream head**.

[EAGAIN] Unable to allocate STREAMS storage to perform the **I_LINK**.

[EBADF] **Arg** is not a valid, open file descriptor.

[EINVAL] **Fildes stream** does not support multiplexing.

[EINVAL] **Arg** is not a **stream** or is already linked under a multiplexer.

[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_LINK** 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 acknowledgment message. For these cases, **I_LINK** will fail with **errno** set to the value in the message.

I_UNLINK Disconnects the two STREAMS specified by **fildes** and **arg**. **Fildes** is the file descriptor of the **stream** connected to the multiplexing driver. **Arg** is the multiplexer ID number that was returned by the **ioctl I_LINK** command when a **stream** was linked below the multiplexing driver. If **arg** is -1, then all STREAMS which were linked to **fildes** are disconnected. As in **I_LINK**, this command requires the multiplexing driver to acknowledge the unlink. On failure, **errno** is set to one of the following values:

[ENXIO] Hangup received on **fildes**.

[ETIME] Time out before acknowledgment message was received at **stream head**.

[EAGAIN] Unable to allocate buffers for the acknowledgment message.

[EINVAL] Invalid multiplexer ID number.

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 acknowledgment message. For these cases, **I_UNLINK** will fail with **errno** set to the value in the message.

RETURN VALUE

Unless specified otherwise above, the return value from **ioctl** is 0 upon success and -1 upon failure with **errno** set as indicated.

SEE ALSO

CLOSE(BA_OS), FCNTL(BA_OS), IOCTL(BA_OS), OPEN(BA_OS), READ(BA_OS), GETMSG(NS_OS), POLL(NS_OS), PUTMSG(NS_OS), SIGNAL(BA_OS), SIGSET(BA_OS), WRITE(BA_OS).

LEVEL

Level 1.

NAME

getmsg - receive next message off a stream

SYNOPSIS

#include <stropts.h>

```
int getmsg(fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;
```

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.

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 */
```

where **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 is 0 if there is a zero-length control or data part; or is -1 if no data or control information is present in the message. **Flags** may be set to the values 0 or **RS_HIPRI** and is used as described below.

Ctlptr is used to hold the control part of the message, and **dataptr** is used to hold the data part of 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, and **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 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 below under *RETURN VALUE.* If information is retrieved from a priority message, **flags** is set to **RS_HIPRI** on return.

By default, **getmsg** processes the first priority or non-priority message available on the **stream head** read queue. However, a user may choose to retrieve only priority messages by setting **flags** to **RS_HIPRI**. In this case, **getmsg** will only process the next message if it is a priority message.

If **O_NDELAY** has not been set, **getmsg** blocks until a message of the type(s) specified by **flags** (priority or either) is available on the **stream head** read queue. If **O_NDELAY** has been set and a message of the specified type(s) 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** will continue to operate normally, as described above, until the **stream head** read queue is empty. Thereafter, it will return 0 in the **len** fields of **ctlptr** and **dataptr**.

ERRORS

Getmsg fails if one or more of the following are true:

[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**, or **flags** 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 **flags**, or the **stream** referenced by **fd** is linked under a multiplexer.

[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.

RETURN VALUE

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of

MORECTL indicates that more control information is waiting for retrieval. A return value of **MOREDATA** indicates that more data is waiting for retrieval. A return value of **MORECTL|MOREDATA** indicates that both types of information remain. Subsequent **getmsg** calls will retrieve the remainder of the message.

SEE ALSO

READ(BA_OS), POLL(NS_OS), PUTMSG(NS_OS), STREAMS(NS_DEV), WRITE(BA_OS)

LEVEL

Level 1.

NAME

poll - STREAMS input/output multiplexing

SYNOPSIS

#include <stropts.h>
#include <poll.h>

int poll(fds, nfds, timeout)
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 that reference open STREAMS. **Poll** identifies those STREAMS on which a user can send or receive messages, or on which certain events have occurred. A user can receive messages using READ(BA_OS) and GETMSG(NS_OS) and send messages using WRITE(BA_OS) and PUTMSG(NS_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 any combination of the following event flags:

POLLIN A non-priority message is present on the **stream** head read queue. This flag is set even if the message is of zero length. In **revents**, this flag is mutually exclusive with **POLLPRI**.

POLLPRI A priority message is present in the **stream** head read queue. This flag is set even if the message is of zero length. In **revents**, this flag is mutually exclusive with **POLLIN**.

POLLOUT The first downstream write queue in the **stream** is not full. Priority messages can be sent [see PUTMSG(NS_OS)] at any time.

POLLERR An error message has arrived at the **stream** head. This flag is only valid in the **revents** bitmask; it is not used in the **events** field.

POLLHUP A hangup has occurred on the **stream**. This event and **POLLOUT** are mutually exclusive; a **stream** can never be writable if a hangup has occurred. However, this event and **POLLIN** 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 does not belong to an open **stream**. 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 **nfds** exceeds **NOFILES**, which is the system limit of open files, **poll** will fail.

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 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 when 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** msec 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** flag.

ERRORS

Poll fails if one or more of the following are true:

[EAGAIN] Allocation of internal data structures failed but request should be attempted again.

[EFAULT] Some argument points outside the allocated address space.

[EINTR] A signal was caught during the **poll** system call.

[EINVAL] The argument **nfds** is less than zero, or **nfds** is greater than **NOFILES**.

RETURN VALUE

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has 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, a value of -1 is returned and **errno** is set to indicate the error.

SEE ALSO

READ(BA_OS), GETMSG(NS_OS), PUTMSG(NS_OS), STREAMS(NS_DEV), WRITE(BA_OS).

LEVEL

Level 1.

NAME

putmsg - send a message on a stream

SYNOPSIS

```
#include <stropts.h>

int putmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
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.

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  */
char *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(NS_OS)]. In a similar manner, **dataptr** specifies the data, if any, to be included in the message. **Flags** may be set to the values 0 or **RS_HIPRI** and is used as described below.

To send the data part of a message, **dataptr** must be non-NULL and the **len** field of **dataptr** must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for **ctlptr**. No data (control) part will be sent if either **dataptr (ctlptr)** is NULL or the **len** field of **dataptr (ctlptr)** is set to -1.

If a control part is specified, and **flags** is set to **RS_HIPRI**, a priority message is sent. If **flags** is set to 0, a non-priority message is sent. If no control part is specified, and **flags** is set to **RS_HIPRI**, **putmsg** fails and sets

errno to **EINVAL**. If no control part and no data part are specified, and **flags** is set to 0, no message is sent, and 0 is returned.

For non-priority messages, **putmsg** will block if the **stream** write queue is full due to internal flow control conditions. For priority messages, **putmsg** does not block on this condition. For non-priority messages, **putmsg** does not block when the write queue is full and **O_NDELAY** is set. Instead, it fails and sets **errno** to EAGAIN.

Putmsg 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_NDELAY** has been specified. No partial message is sent.

ERRORS

Putmsg fails if one or more of the following are true:

[EAGAIN] A non-priority message was specified, the **O_NDELAY** 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.

[EBADF] **Fd** is not a valid file descriptor open for writing.

[EFAULT] **Ctlptr** or **dataptr** points outside the allocated address space.

[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 multiplexer.

[ENOSTR] A **stream** is not associated with **fd**.

[ENXIO] A hangup condition was generated downstream for the specified **stream**.

[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.

A **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.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

SEE ALSO

READ(BA_OS), GETMSG(NS_OS), POLL(NS_OS), STREAMS(NS_DEV), WRITE(BA_OS).

LEVEL

Level 1.

Chapter 15
Shared Resource Environment

15.1 INTRODUCTION

The SHARED RESOURCE ENVIRONMENT section of the NETWORK SERVICES EXTENSION describes a set of capabilities for sharing and administering resources among interconnected machines, that are collectively called *Remote File Sharing*. Using Remote File Sharing, files that physically reside on a remote machine can be accessed as if they were on the local machine; the capabilities described here provide the interface for accessing and managing Remote File Sharing. New utilities provide the basic functionality, while additional functionality is added to the BASE, the BASIC UTILITIES EXTENSION, and the ADMINISTERED SYSTEM EXTENSION.

This extension is dependent on the Base System and on the MOUNT(AS_CMD) and UMOUNT(AS_CMD) utilities in the ADMINISTERED SYSTEM EXTENSION.

15.2 DESCRIPTION

UTILITIES

adv	**idload**	**rfstart**
dname	**nsquery**	**rfstop**
fumount	**rfadmin**	**rmnstat**
fusage	**rfpasswd**	**unadv**

ERROR CONDITIONS

ECOMM	Communications error
EMULTIHOP	Multihop not allowed
ENOLINK	The link has been severed
EREMOTE	The object is remote

15.3 DEFINITIONS

Advertise

Make a directory available as a remotely sharable resource.

Client

A host that has mounted sharable resources from another host (the server).

Domain

An administrative structure for managing the names of a set of hosts and the names of their sharable resources.

Domain Name Server

A host that has the responsibility for maintaining the name space for one or more domains. For each domain, one host is designated the *primary* domain name server, and some number of other hosts are designated *secondary* domain name servers. The host exercising the domain name server responsibility at any given time is the *acting* domain name server. A host need not be a member of a domain for which it is a name server.

Host

A computer running the Remote File Sharing facility. A host may be a client, a server, or both. A host may be a member of only one domain.

Implementation-specific Constants

In addition to the values listed under **Implementation-specific constants** in **Volume I: Part II – Base System Definition: Chapter 4 – Definitions,** several values are defined here.

{NS_RECOVER}	Maximum number of minutes before domain name service recovery.
{RDESC_MAX}	Maximum number of characters in a resource description.

Multihop Access

Multihop access refers to the following structure of "indirect" access to a remote resource. Suppose host A advertises a resource that has mounted within it a resource from host B. If any other host mounts the resource from host A and uses it to access a file on the resource from host B, then that access is termed *multihop* access.

Server

A host that has one or more advertised resources.

Sharable Resource

A sharable resource is one that is advertised by a server and thus is available to authorized clients.

15.4 ERROR CODES

In addition to the error codes defined as part of the SOFTWARE DEVELOPMENT EXTENSION, new error codes have been added to the header file **errno.h.**

<errno.h>

Defines the following symbolic names for the indicated error return conditions:

Name	*Description*
ECOMM	Communications error
EMULTIHOP	Multihop not allowed
ENOLINK	The link has been severed
EREMOTE	The object is remote

The ECOMM error condition occurs on any operating system service routine that references a remote resource (through a file descriptor or path name), whenever there is a communications error while trying to send the request for that service routine to the server machine. The EMULTIHOP error condition may occur on any operating system service routine that has a path name as one of its arguments, and indicates that resolution of that path name involves multihop access to a remote resource, when multihop access is not supported by the underlying implementation. Whether multihop access is supported is implementation-specific, but if it is not supported, then the EMULTIHOP error condition must be returned on any attempted multihop access. The ENOLINK error condition occurs on any operating system service routine that references a remote file, when the communications link to the server for that resource has been lost; any file descriptor associated with this remote file should not be used for further I/O.

The EREMOTE error condition occurs on the MOUNT(BA_OS) operating system service routine when the requested mount point resides on a remote resource.

15.5 EFFECTS ON THE BASE SYSTEM

Header Files

Under the Shared Resource Environment, components in the Base System may return a new value for **errno**, as listed above. In addition, some operating system service routines may return the **errno** value of **EINTR** when accessing a remote resource.
The operating system service routines that do not return this value of **errno** except under the Shared Resource Environment are:

access	**creat**	**mknod**
chdir	**dup**	**stat**
chmod	**exec**	**unlink**
chown	**fcntl**	**ustat**
close	**link**	**utime**

An application that checks the value of **errno** must include the header file <**errno.h**>.

15.6 EFFECTS ON OTHER EXTENSIONS

Some of the utilities in other Extensions are affected by the additional services in the Network Services Extension. The effects are listed below for each utility within each affected Extension.

15.6.1 Effects on the BASIC UTILITIES EXTENSION

Df(BU_CMD)

Df(BU_CMD) is updated to provide free block and free inode information on remote resources mounted locally in addition to local resources. The new syntax is:

df [-l] [-t] [*file_system* | *resource*] ...

When used with no arguments, df(BU_CMD) previously reported on all mounted file systems; now df(BU_CMD) will report on both mounted file systems and mounted remote resources. If the **–l** option is used, **df** will report only on the local file systems.

If the mounted remote resource is a file system, the free space data are reported for the remote file system. If the mounted remote resource is not a file system, free space data for the parent file system are reported; when df(BU_CMD) reports on more than one resource from a specific file system, the second and subsequent entries in the report will be flagged by an asterisk.

Find(BU_CMD)

Find(BU_CMD) is updated to be able to distinguish local and remote files. A new primary is defined, **–local**, which is true if the file physically resides on the local system.

15.6.2 Effects on the ADMINISTERED SYSTEM EXTENSION

Mount(AS_CMD)

When the Network Services Extension is present, a **–d** option is available with the MOUNT(AS_CMD) utility. This option is used to locally mount remote resources that have been advertised by a server. The complete syntax is:

mount [[–d] [–r] *special directory* **]**

When used with no options, MOUNT(AS_CMD) will report on both local file systems and remote resources that have been mounted.

When the **–d** option is used, *special* must be a valid resource name of the form *resource* or *domain.resource.* For users and applications processes, the effect of a remote mount is the same as a local mount: an additional file system has been mounted into the local file tree. Once a remote resource has been mounted, all operating system service routines will operate on the remote files as they do on local files, except that it is implementation-specific whether the following operating system service routines will accept a remote file:

acct(KE_OS)	**poll(NS_OS)**
getmsg(NS_OS)	**putmsg(NS_OS)**

Future Direction

The four operating system service routines listed above will be extended in the future to operate with remote files.

Errors

If the **–d** option is used and (1) Remote File Sharing is not running on this host, (2) the mount point *directory* is itself advertised as a resource, (3) the mount point *directory* is already a mount point, (4) the **–r** is not specified and the resource was advertised as read-only, (5) the resource is not currently advertised,

(6) the resource is already mounted, or (7) the client is not authorized to access the resource, an error message will be sent to standard error.

Umount(AS_CMD)

When the Network Services Extension is present, an additional option, **–d**, is available with UMOUNT(AS_CMD) for unmounting remote resources mounted locally. The complete syntax is:

umount [**–d**] *special*

If the **–d** option is used, *special* must be a valid resource name of the form *resource* or *domain.resource*.

Errors

Additional error conditions can arise on servers when they attempt to unmount local resources that are currently advertised or remotely mounted. If (1) the resource has not been unadvertised or (2) the resource is still currently mounted on a remote machine, an error message will be sent to standard error.

Fuser(AS_CMD)

There are no changes to the syntax for FUSER(AS_CMD), but remote resources mounted locally can now be specified on the command line by giving the resource name as an argument. Although a local file can still be used as an argument, the command will issue a warning if a remote file is specified.

Sar(AS_CMD)

When the Network Services Extension is present, the options **–S** and **–D** are available with SAR(AS_CMD). If neither of these options are specified on the command line, the output of SAR(AS_CMD) will not change. The complete syntax is:

sar [**–ubdycwaqvmprADS**] [**–o** *file*] **t** [**n**]
sar [**–ubdycwaqvmprADS**] [**–s** *time*] [**–e** *time*] [**–i** *sec*] [**–f** *file*]

The **–D** option is used in combination with either the **–u** or **–c** option. If the **–D** is used and neither **–u** nor **–c** is specified, **–u** is assumed.

The command **sar -u** reports time spent in user mode, in system mode, idle with some process waiting for block I/O, and otherwise idle. If the **–D** option is also specified, system time is reported for time servicing remote requests and all other system time. The command **sar -c** reports activity data on system calls. If the **–D** option is also specified, the data are reported for three categories: system calls resulting in outgoing remote activity, system calls resulting from incoming remote activity, and strictly local system calls.

The **–S** option is used to obtain reports on server processes and request queue status. Every request from a remote host to access your resources is conveyed by a request message that is handled by a *server process.* When there are too many messages for the servers to handle, the messages are placed on a request queue. Messages leave the queue and are processed when servers are available. The data reported by the **–S** option are the following: average number of server processes on the system (**serv/lo-hi**), % of time request messages are on the request queue (**request %busy**), average number of request messages waiting for service when the request queue is occupied (**request avg lgth**), % of time there are idle servers (**server %avail**), and average number of idle servers when idle ones exist (**server avg avail**).

Sa1(AS_CMD)

The new **–S** and **–D** options described for SAR(AS_CMD) are also available for **sa2**; the interfaces to **sa1** and **sadc** are unchanged. The complete syntax for **sa2** is:

/usr/lib/sa/sa2 [**–ubdycwaqvmprADS**] [**–s** *time*] [**–e** *time*] [**–i** *sec*]

15.7 CONFORMING SYSTEM CHARACTERISTICS

This section delineates characteristics that all systems must possess in order to conform to the Shared Resource Environment. From an application perspective, these are characteristics of the overall Shared Resource Environment, and do not reside in any one component. Thus, all conforming systems will have the following characteristics, in addition to the individual component interfaces presented in the SHARED RESOURCE ENVIRONMENT section of the NETWORK SERVICES EXTENSION, in order to ensure portability of source code from single-machine environments to a network of machines sharing resources.

15.7.1 Network Compatibility

There are implementation-specific criteria for what underlying network(s) can be used to support the Remote File Sharing capabilities described in the SHARED RESOURCE ENVIRONMENT section of the NETWORK SERVICES EXTENSION, but if two machines can each use a given network to support Remote File Sharing with some other machines, then they will be able to jointly engage Remote File Sharing with each other (using that network).

15.7.2 Operation Across Heterogeneous Processors

Some application-level operations may depend on characteristics of the underlying processor. For example, when an application writes a floating-point number into a file, it is typically stored in a format specific to that processor, which may differ in size or byte-ordering from the representation of the same number on a different processor. Similar considerations apply to the representation of more elaborate structured data items, which may also differ across processors in their alignment characteristics. Because the identification and interpretation of such complex data items is solely under the control of the application process, and is not known to the operating system, the operating system cannot automatically perform the translations required for the proper interpretation of those data items when they are shared among processors of different types.

However, for any set of machines that are able to engage in Remote File Sharing with one another, applications on those machines will be able to share named pipes (FIFO's) and any files that are regarded purely as a sequence of bytes (such as ASCII files) without concern for the underlying processor characteristics. Furthermore, by agreeing on a standard external data representation format, applications may manipulate arbitrarily complex data items as a pure sequence of bytes, and thus share those data items across dissimilar processors.

15.7.3 Reliability Against a Single Point of Failure

If a machine that conforms to the SHARED RESOURCE ENVIRONMENT section of the NETWORK SERVICES EXTENSION turns off its Remote File Sharing facility, it must not cause domain name service to be halted completely; service may be interrupted, however, for up to {NS_RECOVER} minutes. Within this time interval, domain name service must be resumed, even if the departing machine has not resumed Remote File Sharing. During the outage interval, new MOUNT(AS_CMD) requests do not have to be honored, but previously mounted resources must continue to work as before, unless they physically reside on the machine that stopped its Remote File Sharing facility. Information maintained by the domain name service (such as the list of currently advertised resources) must be retained across the outage.

15.8 FILE SHARING

The Remote File Sharing facility of the Shared Resource Environment provides access to files from a remote machine as though they were on the machine you are logged into. Remote files are named using the same conventions as for local files, and all operations on remote files work the same as they do on local files.

This section presents an overview of the functionality and administrative features of Remote File Sharing. It is included as background for understanding this part of the Network Services Extension of the System V Interface Definition.

Every machine participating in Remote File Sharing is able to make selected parts of its file tree available for sharing, by *advertising* them. Correspondingly, each machine is able to augment its own file tree with the advertised files from other machines. This augmentation is performed by means of a *remote mount*, which is a direct extension of the standard *mount* operation. This section describes the *advertise*, *unadvertise*, and *remote mount* concepts.

15.8.1 Advertise

The right to allow file sharing belongs to the administrator of the machine where the file resides. To allow sharing, an administrator *advertise*s a directory using the **adv** command. Once advertised, the directory and all files in the subtree below it, including named pipes and special devices, are available for sharing by any authorized machine. (How a machine becomes "authorized" is discussed later.)

15.8.2 Unadvertise

The administrator can *unadvertise* a directory at any time after it has been advertised by using the **unadv** command. Unadvertising a directory has no effect on existing mounts of the directory, but future mount requests will fail.

15.8.3 Remote Mount

The Shared Resource Environment extends the MOUNT(AS_CMD) operation to include a *remote mount*. After a machine has advertised a resource, another machine may remotely mount that resource in its own file tree.

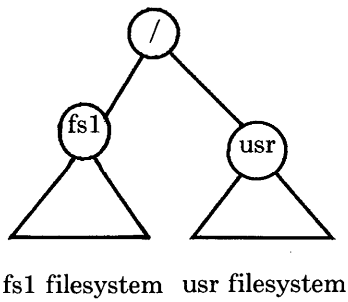

Figure 15-1: A System V File System

Figure 15-1 shows part of a typical file tree. To advertise the subtree under **/fs1**, you type

```
adv DATA /fs1
```

This makes the **/fs1** subtree available for sharing, and specifies that other machines will use the name **DATA** to refer to it when they mount it. The name **DATA** can be almost any name that would work as a file name as long as it does not contain a period ("."). The period has a special meaning that will be discussed later.

Another machine gains access to the advertised subtree by mounting the remote subtree on a local directory. An administrator mounts the remote **/fs1** subtree advertised above on the local **/fs1** directory by typing

```
mount -d DATA /fs1
```

The **-d** option tells the **mount** command that the resource being mounted is remote.

The machine that owns a file is called the *server machine*, while the machine that uses the file is called the *client machine*.

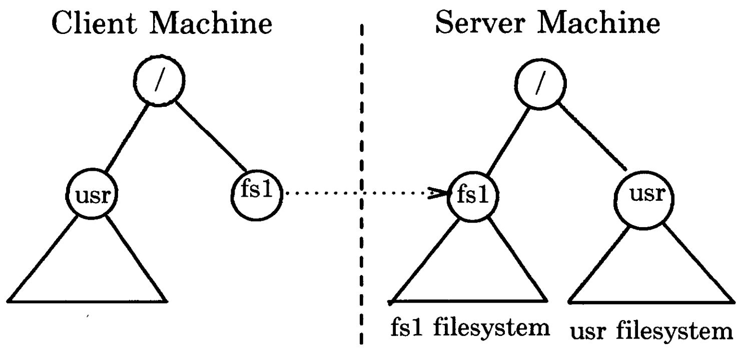

Figure 15-2: Remote Mount

Figure 15-2 shows the two machines' file systems after the remote mount. The dotted line connecting the directories means that when a user on the client refers to the subtree under **/fs1**, the file referenced is the one on the server subtree under **/fs1**. For example, a user on the client machine who uses the file name **/fs1/src/uts** refers to the file by that name on the server machine.

There is no need for the structures of client and server file trees to match in any way, or for advertised subtrees to be mounted at the same level on the client as they occupy on the server. If the client had done the remote mount onto its **/usr** directory, then its references to files under **/usr** would be to the server subtree under **/fs1**.

A client cannot get to parts of the server file tree that are not under the advertised directory. For example, if a user on a client machine uses "cd .." to move up from the top directory in a remotely mounted subtree, the user always ends up back in the client file tree.

15.9 ADMINISTRATIVE FEATURES

This section describes the resource naming and security features of Remote File Sharing.

15.9.1 Resource Naming

Resource naming is modeled after the proposed ARPA domain naming convention[1], which has a hierarchically structured name space. A domain in this usage is a name space that may encompass a group of machines and a set of resources advertised by that group of machines.

Resource names are made up of two components separated by a period (.). For example, *isl.payroll* might represent a resource called *payroll* in domain *isl*, and *isl.acctp* might represent the machine *acctp* within that same domain. Whether a name specifies a resource or a machine is determined by context; there is no syntactic distinction. If a name is unqualified (i.e., contains no periods), the associated domain may (in some cases) be inferred from the context.

A domain's name space is maintained by a *domain name server*, which insures uniqueness of names within the domain and provides a central place for storing information about the machines and advertised resources in the domain. The ADV(NS_CMD), UNADV(NS_CMD), MOUNT(AS_CMD), UMOUNT(AS_CMD), and NSQUERY(NS_CMD) commands use the domain name server as a data base for information about advertised resources, such as their names and the servers that own them. The Network Services Extension of the System V Interface Definition defines the interface to this name service through new administrative utilities.

As described above, each resource is assigned a symbolic name when it is advertised, and the resource is subsequently identified (say, within a MOUNT(AS_CMD) command issued on a client) using just the domain name and that symbolic name. Because of this symbolic naming of resources, administrators of other machines need not know the actual position of the resource within the server's file tree, nor even what server within the domain is offering the resource. This location independence simplifies references to resources, and allows for the transparent migration of resources among the machines within a domain (for example, for balancing the load among a set of server machines).

15.9.2 Future Directions

In the future, the domain name space will be extended to include subdomains. By allowing a domain to include subdomains, a tree-structured naming hierarchy can be built, to ease adminstration of large numbers of resources.

15.9.3 Security Features

There are three features that provide a more secure environment for sharing files: client authentication, client authorization, and user and group id mapping. Most administrators want to be sure that access to files is provided only to those clients that are known and, to at least some degree, trusted. In addition, they normally want to have control over the user and group ids that remote users have on their machines.

Client Authentication

This feature associates a password with a client machine so that the identity of the prospective client can be checked before a mount request is serviced. Entry and update of passwords is discussed in the sections on the RFADMIN(NS_CMD), RFSTART(NS_CMD), and RFPASSWD(NS_CMD) commands.

Client Authorization

The Remote File Sharing facility provides a way for an administrator to selectively advertise directories through the ADV(NS_CMD) command. For example, if you want to advertise **/usr/private**, but only want to authorize machines *mach1* and *mach2* to mount that directory, you would issue the command:

```
adv PRIVATE /usr/private mach1 mach2
```

Without a list of machines, the ADV(NS_CMD) command puts no restriction on availability.

An administrator may also choose to advertise a directory read-only by using the **−r** option. Here, a remote mount will only succeed if the mount command also includes the **−r** option.

User and Group Id Mapping

Whenever a user accesses a remote file, that user's permissions must be checked as part of the normal processing of the request (for example, an "open to write" is only valid if the user making the request has write permissions on the file). When accessing a file across two machines, there is no guarantee that the user and group ids on the local machine have the same meaning on the other machine.

Some systems handle this problem by requiring the same numeric ids across machines and expecting the administrators to make sure that the **/etc/passwd** and **/etc/group** files are identical across all machines (at least the entries for all users that access remote files). This approach is very straightforward from the perspective of administrative simplicity, but it is not always feasible, especially in large or already established environments.

Remote File Sharing provides a range of id mapping options through the IDLOAD(NS_CMD) command. Id mapping is done by a server machine on all incoming requests, as well as in reporting file ownership ids in response to a request from a client machine (for example, STAT(BA_OS) and FSTAT(BA_OS). A client machine maps ids in order to determine the effective user or group id to use in executing a program that is stored on a server and is "set user id" or "set group id".

On each machine, mapping can be set globally, for all remote machines, or on a per-machine basis. All mapping is based on one of two default cases:

id — This case maps all incoming ids to **id**, which means that remote users will have the permissions associated with **id in** accessing a server's files. This mapping is the default if no other mapping is specified.

transparent — This is a null mapping; remote user and group ids are used locally without change.

These base mappings are augmented by two additional capabilities:

exclude — This capability excludes selected ids from the default mapping by mapping them to an otherwise-unused id. This capability could be used together with the transparent mapping capability to handle a network where the **/etc/passwd** and **/etc/group** files were identical, but the administrators did not want to allow certain permissions (for example, root) from remote machines.

map — This capability provides arbitrary mapping between remote and local ids that have different names or different numeric values. It could be used with the transparent mapping to handle exceptions to "nearly" identical **/etc/passwd** files.

15.9.4 References

[1] Su, Z. S., and J. Postel, *The Domain Naming Convention for Internet User Applications" , RFC-819, Network Information Center, SRI International, August, 1982.*

NAME

adv - advertise a directory for remote access

SYNOPSIS

adv [**−r**] [**−d** *description*] *resource pathname* [*client* ...]
adv **−m** *resource* **−d** *description* [*client* ...]
adv **−m** *resource* [**−d** *description*] *client* ...
adv

DESCRIPTION

The **adv** command is used to make a resource from one computer available for use on other computers. The machine that advertises the resource is called the *server*, while computers that mount and use the resource are *clients*. A resource is composed of a directory and everything under that directory (including subdirectories); the directory must be within a file system that is mounted locally.

There are three ways **adv** is used: (1) to advertise the directory *pathname* under the name *resource* so it is available to each *client*; (2) to modify *client* and *description* fields for currently advertised resources; or (3) to print a list of all locally-advertised resources.

The following options are available:

−r Restricts access to the resource to a read-only basis. The default is read-write access. If the resource is on a read-only file system, it must be advertised read-only, while resources with local read-write access may be advertised either read-only or read-write.

−d *description* Provides brief textual information about the advertised resource. The *description* is a single argument surrounded by double quotes (") and has a maximum length of {RDESC_MAX} characters; a description longer than {RDESC_MAX} characters will be truncated and result in a warning message.

resource This is the symbolic name used by the server and all authorized clients to identify the resource. The *resource* name can be up to a maximum of {NAME_MAX} characters long and must be different from every other resource name in the domain; a resource name longer than {NAME_MAX} characters will be truncated and result in a warning message. All characters must be printable ASCII characters but must not include periods (.), slashes (/), or white space.

pathname — This is the absolute path name of the advertised resource on the local host. The *pathname* cannot be the mount point of a remote resource and it can only be advertised under one resource name.

client — This specifies client machines that are authorized to remotely mount the resource. If no *client* is named, all machines that can connect to the server are authorized to access the resource. *client* is of the form *nodename*, *domain.nodename*, *domain.*, or an alias that represents a list of client names. A domain name must be followed by a period (.) to distinguish it from a host name. The aliases are defined in **/etc/host.alias**; the syntax of this file matches the alias capability in MAILX(AU_CMD).

−m *resource* — This option modifies information for a resource that has already been advertised. The resource is identified by a *resource* name. Only the *client* list and the *description* field can be modified with this option.

When used with no options, **adv** displays all local resources that have been advertised; this includes the resource name, the path name, the description, the read-write status, and the list of authorized clients. The resource field has a fixed length of {NAME_MAX} characters; all others are of variable length. Fields are separated by two spaces and double quotes (") surround the description.

This command may be used without options by any user; otherwise, it is restricted to the super-user.

The host must be running Remote File Sharing before **adv** can be used to advertise or modify a resource entry.

ERRORS

If there is at least one syntactically valid entry in the *client* list, and the command is otherwise valid, a warning will be issued for each invalid entry and the command will return a zero exit status.

If (1) the resource name is not unique within the server's set of advertised resources and the **–m** option is not used, (2) the resource name is not unique within the host's domain, (3) *resource* is not a directory, (4) the resource is not on a file system mounted locally, (5) there is at least one *client* specified but none are syntactically valid, (6) the **–m** option is used but neither a description nor a client list is provided, or (7) the resource name contains a period (.) or a slash (/), an error message will be sent to standard error.

FILES

/etc/host.alias

USAGE

Administrator, End-User.

SEE ALSO

MOUNT(NS_CMD), RFSTART(NS_CMD), UNADV(NS_CMD)

LEVEL

Level 1

NAME

dname - print or set the domain name of the host

SYNOPSIS

dname [**–D** *domain*] [**–d**]

DESCRIPTION

Without options, or when used with the **–d** option, **dname** prints the name of the domain. When used with the **–D** option, **dname** changes the domain name of the host to *domain*.

The *domain* name must be from 1 to {NAME_MAX} characters, consisting of any combination of letters (upper and lower case), digits, hyphens (-), and underscores (_). If *domain* is not a valid domain name, the previous name is retained.

The domain name cannot be changed while Remote File Sharing is running.

Any user can execute this command without options or with the **–d** option, but only the super-user can use the **–D** option to set the domain name.

ERRORS

When **dname** is used without options and (1) the domain name is not set or (2) the domain name cannot be accessed, an error message will be sent to standard error.

When the **–D** option is used and (1) the user is not the super-user, (2) Remote File Sharing is running, or (3) the new domain name is not syntactically valid, an error message will be sent to standard error.

USAGE

Administrator, End-User.

SEE ALSO

NSQUERY(NS_CMD), RFSTART(NS_CMD)

LEVEL

Level 1

NAME

fumount - forced unmount of an advertised resource

SYNOPSIS

fumount [**-w** *sec*] *resource*

DESCRIPTION

The **fumount** command unadvertises the specified *resource*, triggers a remote warning to clients that have the *resource* mounted, and disables remote access to the *resource*. The **-w** *sec* causes a delay of *sec* seconds prior to the execution of the disconnect.

When **fumount** sends its warning to remote clients, as well as when it actually disables remote access to a resource, it triggers the execution of an administrative shell script on the remote system(s). This shell script can be modified by the administrator of each machine, in order to customize the actions taken in response to such **fumount** operations. The location and default actions of this shell script are implementation-dependent.

The **fumount** command issues a warning message if the resource is remotely mounted but is not advertised.

This command is restricted to the super-user.

ERRORS

If *resource* (1) does not physically reside on the local machine, (2) is an invalid resource name, (3) is not currently advertised and is not remotely mounted, or if (4) the command is not run with super-user privileges, an error message will be sent to standard error.

USAGE

Administrator.

SEE ALSO

ADV(NS_CMD), MOUNT(AS_CMD), UMOUNT(AS_CMD), UNADV(NS_CMD)

LEVEL

Level 1.

NAME

fusage - disk access profiler

SYNOPSIS

fusage [*mount_point* ¦ *advertised_resource* ¦ *blk_special_dev*] ...

DESCRIPTION

When used with no options, **fusage** reports block i/o transfers, in kilobytes, to and from all locally mounted file systems and advertised resources on a per client basis. The count data are cumulative since the time of the mount. When used with an option, **fusage** reports on the named file system, advertised resource, or block special device.

The report includes one section for each file system and advertised resource and has one entry for each machine that has the directory mounted, ordered by decreasing usage. Sections are ordered by device name; advertised resources that are not complete file systems will immediately follow the sections for the file systems they are in.

USAGE

Administrator, End-User.

SEE ALSO

ADV(NS_CMD), MOUNT(AS_CMD)

LEVEL

Level 1.

NAME

idload - user and group ID mapping

SYNOPSIS

idload [**–n**] [**–g** *g_rules*] [**–u** *u_rules*] [*directory*]

DESCRIPTION

The **idload** command is used to build translation tables for user and group ids. These tables are used to translate between the user and group id of a user on a client machine and the ids on a server machine. When a server is responding to a request from a user on a client, the server uses its tables to decide what permissions to give the client user. A server also uses its tables in reporting file ownership ids for its own files, translating them into the corresponding ids for the client. A client uses its tables only when executing a set-uid or set-gid program that is stored on a server; it uses its translation tables to decide what effective user or group id to give the program when it executes on the client.

The **idload** command produces the user and group translation tables according to the rules set down in the *u_rules* and *g_rules* files. If the rules files are empty, or if **idload** has never been run, remote user and group ids are mapped to the value {UID_MAX}+1. [An id of {UID_MAX}+1 is just like a regular user or group id, except that no local user can be assigned this id.]

Ids are always mapped within the system by their numeric value, but the rules files given to **idload** can use both names and numeric ids to describe the mapping. If any local users or groups are specified by name in the rules files, **idload** uses the local **/etc/passwd** and **/etc/group** files to translate those names into numeric ids. If any remote users or groups are specified by name, instead of by numeric id, **idload** must be able to access copies of the **/etc/passwd** and **/etc/group** files from the corresponding remote hosts. **idload** looks for the remote password and group files under the names *directory/domain/host/***passwd** and *directory/domain/host/***group**, respectively, where *directory* is taken from the command line, and *domain* and *host* specify the desired host (*domain.host*). If *directory* is not specified, the default value of **/usr/nserve/auth.info** is used.

The following options can be used with **idload**:

–n No change will be made to the translation tables currently in effect. Instead, **idload** will interpret the input files, and will print a description of the results on standard output.

–u *u_rules* The *u_rules* file contains the rules for user id translation. ("If the **-u** option is omitted, the default rules file **/usr/nserve/authinfo/uid.rules** is used.")

–g *g_rules* The *g_rules* file contains the rules for group id translation. ("If the **-g** option is omitted, the default rules file **/usr/nserve/authinfo/gid.rules** is used.")

This command is restricted to the super-user.

A host need not be running Remote File Sharing in order to run **idload**; the new mapping will take effect when the host next starts Remote File Sharing. If the host is running Remote File Sharing when **idload** is run successfully, the new mapping takes effect immediately.

RULES FILES

The *u_rules* and *g_rules* files are built according to the same syntax rules. The following text describes the mapping of user ids and user names in a *u_rules* file; the method for mapping group ids in a *g_rules* file is directly parallel. Note that the group id mapping is completely independent of the user id mapping.

The rules file has two types of sections, both optional: **global** and **host**. There can be only one **global** section, but there can be as many **host** sections as needed.

The **global** section describes the default conditions for translation of user ids from any machines that are not explicitly referenced in a **host** section. If the **global** section is missing, the default action is to map all remote user ids from unspecified hosts to the value {UID_MAX}+1.

A **host** section is used for each host or group of hosts that is to be mapped differently from the global definitions. The first line of a **host** section names the host(s) that are described by that section. If multiple hosts are described in a single **host** section, and any users (or groups) are specified by name, **idload** will read the **passwd** (or **group**) file only for the first host named and will use that information for all hosts in the section. A host can only be described once within any one rules file.

The overall format of a rules file is described below. Each of the instructions listed within the **global** and **host** sections is optional, but, if used, must appear in the order shown. Following this overall format is an explanation of each of the individual instruction types.

global
default *local* ¦ **transparent**
exclude *remote_id–remote_id* ¦ *remote_id ...*
map *remote_id*:*local* ¦ *remote_id ...*

\# comment text
host *domain.host ...*
default *local* ¦ **transparent**
exclude *remote_id–remote_id* ¦ *remote_id* ¦ *remote_name ...*
map *remote*:*local* ¦ *remote ...*
map all

Any line beginning with a '#' is regarded as a comment, and is otherwise ignored by **idload.**

The line

default *local* ¦ **transparent**

defines the mode of mapping for remote users that are not specifically mapped in other instructions. *local* can be replaced by a local user name or id to map all remote users into a particular local name or id number. The default value cannot be root; thus, *local* can be neither **0** nor **root**. **transparent** means that each remote user id will have the same numeric value locally unless it appears in the **exclude** or **map** instruction. If the **default** line is omitted, all remote ids that are not explicitly mapped with the **map** instruction are mapped to the value {UID_MAX}+1.

The line

exclude *remote_id–remote_id* ¦ *remote_id* ¦ *remote_name ...*

defines remote numeric id(s) and remote names that should be excluded from the default mapping, and that will instead be mapped to the value of {UID_MAX}+1. Each item in the list to be excluded can be a range of numeric ids, a single numeric id, or a single name (although a *remote_name* cannot be used in a **global** section). This instruction may be repeated as many times as needed.

The lines

> **map** *remote:local* | *remote* ...
>
> **map all**

define the local user ids and names that remote user ids and names will be mapped into. The first form of the **map** instruction is used to map individual ids; this instruction may be repeated as many times as needed. In this instruction, *remote* may be either the login name or the numeric id of a remote user; similarly, *local* may be either the login name or numeric id of a local user. (However, remote names cannot be used in a **global** section.) An id pair *remote:local* says to map the id for that remote user into the id for the local user. If *remote* and *local* are identical within a pair, the *:local* part may be omitted. In the second form of the **map** instruction, the literal entry **all** says to map each of the user names in the remote system's **passwd** file (or **group** file, for group names) into the id for the same name on the local host. (**map all** cannot appear in a **global** section.)

ERRORS

On successful completion, **idload** will modify the user and group mapping currently in effect and will return a zero exit status. If **idload** fails for either type of mapping (user or group), it will return a non-zero exit status without modifying the current mapping of that type.

If any id is mapped more than once within a single **host** or **global** section, a warning will be issued, all mappings for that id but the first will be ignored, and **idload** will continue processing.

If (1) a rules file cannot be found or opened, (2) there are syntax errors in the rules file(s), (3) there are semantic errors in the rules file(s), (4) remote user or group names are used, but the requisite host information could not be found, or (5) the command is not run with super-user privileges, an error message will be sent to standard error.

FILES

/etc/passwd
/etc/group
/usr/nserve/auth.info/*domain/host/***passwd**
/usr/nserve/auth.info/*domain/host/***group**
/usr/nserve/auth.info/uid.rules
/usr/nserve/auth.info/gid.rules

USAGE

Administrator.

If remote users and groups are mapped by name, the requisite **passwd** and **group** files can be gathered on just one host (such as the primary domain name server), advertised as a resource, and then accessed from other hosts by being remotely mounted under *directory*.

EXAMPLE

The following is an example of a **u_rules** file.

```
global
default transparent
exclude 0-100

host music.sonata
exclude fred mary
map all
```

This sample file is composed of a **global** section and one **host** section, for the host *sonata* in domain *music*. For all hosts other than *music.sonata*, the user ids will be mapped transparently, that is, to the identical numerical value on the local host. Excluded from this transparent mapping are ids between 0 and 100, which will instead be mapped to {UID_MAX}+1.

For host *music.sonata*, each user id will be mapped to the one with the same login name on the local host, with the exception of remote users *fred* and *mary*, which will be mapped to {UID_MAX}+1. Those remote user ids that have no matching login name in the local **/etc/passwd** file will also be mapped to {UID_MAX}+1.

SEE ALSO

MOUNT(NS_CMD)

LEVEL

Level 1.

NAME

nsquery - query name server information

SYNOPSIS

nsquery [**-h**] [*name*]

DESCRIPTION

The **nsquery** command provides information about resources available from both the local domain and from other domains. When used with no options, **nsquery** identifies all resources that have been advertised in the local domain. The **-h** option causes header information to be omitted from the display. A report on selected resources can be obtained by specifying a *name*, where *name* is one of the following:

nodename	The report will include only those resources available from the host *nodename*.
domain.	The report will include only those resources available from *domain*.
domain.nodename	The report will include only those resources available from the host *domain.nodename*.

When *name* does not include a period (.), it will be interpreted as a nodename within the local domain. If the *name* ends with a period (.), it will be interpreted as a domain name.

The information contained in the report on each resource includes its advertised name (*resource*), its read/write permissions, the server (*domain.nodename*) that advertised the resource, and a brief textual description of the resource.

FUTURE DIRECTION

In the future, the **nsquery** command will be changed so that if the resource was advertised with a restricted client list that does not include this host, the read/write permissions are listed as **inaccessible**.

ERRORS

If no entries are found when **nsquery** is executed, a zero exit status is returned.

If (1) the domain name server cannot be contacted, or (2) *name* is not known to the domain name server, an error message will be sent to standard error.

USAGE

Administrator, End-User.

SEE ALSO

ADV(NS_CMD), UNADV(NS_CMD)

LEVEL

Level 1

NAME

rfadmin - domain administration

SYNOPSIS

rfadmin
rfadmin –a *hostname*
rfadmin –r *hostname*
rfadmin –p

DESCRIPTION

The **rfadmin** command is used to add and remove hosts and their associated authentication information from the domain membership list(s) maintained on a primary domain name server. It is also used to transfer domain name server responsibilities from one host to another. Used with no options, **rfadmin** prints the name of the current domain name server for the local domain, in the form *domain.nodename.*

The **rfadmin** command can only be used to modify the domain membership lists on the primary domain name server (**–a** and **–r** options). Any host acting as the domain name server can use the **–p** option to pass the domain name server responsibility to another machine. Finally, any host running Remote File Sharing can use **rfadmin** with no options to print the name of the current domain name server. In all cases, the user must have **root** permissions to use the command.

–a *hostname* — Used to add a host to a domain that is served by this domain name server. *hostname* must be of the form *domain.nodename.* It creates an entry for *hostname* in the *domain*/**passwd** file and prompts for an initial authentication password; the password prompting process conforms to that of PASSWD(AU_CMD). The *domain*/**passwd** file has a format similar to **/etc/passwd;** it consists of name and encrypted password fields separated by a colon (:).

–r *hostname* — Used to remove a host from its domain by removing it from the *domain*/passwd file. *Hostname* must be of the form *domain.nodename.*

–p — Used to pass the domain name server responsibilities to another host. The host that will assume the domain name server responsibility is the first one available from a previously-specified list; the list contains the primary name server as the first choice, and some number of other hosts that function as secondary name servers. The means by which a domain administrator specifies this list is implementation-specific.

ERRORS

When used with the **–a** option, if (1) *hostname* is not unique in the domain or (2) the password prompting process fails, an error message will be sent to standard error.

When used with the **–r** option, if (1) *hostname* does not exist in the domain or (2) *hostname* is defined as a domain name server, an error message will be sent to standard error.

If there are no alternative domain name servers defined for *domain* when the **–p** option is used, an error message will be sent to standard error.

If the command is run without super-user privileges, an error message will be sent to standard error.

FILES

/usr/nserve/auth.info/*domain***/passwd**

USAGE

Administrator.

SEE ALSO

PASSWD(AU_CMD), RFPASSWD(NS_CMD), RFSTART(NS_CMD)

LEVEL

Level 1.

NAME

rfpasswd – change host authentication password

SYNOPSIS

rfpasswd

DESCRIPTION

rfpasswd updates the authentication password for a host; processing of the new password follows the same criteria as PASSWD(AU_CMD). The updated password is registered at the domain name server (in **/usr/nserve/auth.info/***domain***/passwd**) and replaces the password stored at the local host.

This command is restricted to the super-user.

ERRORS

If (1) Remote File Sharing is not currently active on this host, (2) the old password entered from this command does not match the existing password for this host, (3) the two new passwords entered from this command do not match, (4) the new password does not satisfy the security criteria in PASSWD(AU_CMD), (5) the domain name server does not know about this host, or (6) the command is not run with super-user privileges, an error message will be sent to standard error.

FILES

/usr/nserve/*domain***/passwd**

USAGE

Administrator.

SEE ALSO

PASSWD(AU_CMD), RFSTART(NS_CMD)

LEVEL

Level 1.

NAME

rfstart - start Remote File Sharing

SYNOPSIS

rfstart [−v] [−p host_addr]

DESCRIPTION

The **rfstart** command starts Remote File Sharing on a host and defines an authentication level for incoming mount requests. **rfstart** supports two levels of host authentication. When executed, **rfstart** always sends a password for this host to the domain name server to authenticate this host's identity. A second level of verification is controlled by an administrator via the **−v** option.

−v Specifies that every client that requests to mount a resource from this host must be verified against the *domain*/**passwd** file; any host for which there is no entry in *domain*/**passwd**, or that does not provide the correct password, will not be allowed to mount resources from this host. If **−v** is not specified, hosts named in *domain*/**passwd** will still be verified, but mount requests from other hosts will be granted without verification. (In the above, *domain* is the domain of the client machine.)

−p *host_addr* Specifies the network address of the domain name server; the syntax for *host_addr* is implementation specific. How the system determines the domain name server when the **−p** option is not used is implementation specific (but see below).

If the host password has not been set, **rfstart** will prompt for a password; the password prompting process conforms to that of LOGIN(AU_CMD). The password entered must match that previously entered on the domain name server for this host with RFADMIN(NS_CMD). If the password entered matches that on the domain name server, it will be set as the local host password. If it does not match, the password will remain unset on the local host, so that **rfstart** will again prompt for the password on its next invocation.

When **rfstart** is executed successfully on a host other than the domain name server, the host will receive from the domain name server the host names and addresses of the primary and secondary name servers for the local domain. The location and format of this information is implementation-dependent, but it will be used by subsequent invocations of **rfstart** in the absence of the **–p** option.

This command is restricted to the super-user.

ERRORS

If (1) Remote File Sharing is already running, (2) there is no communications network, (3) the domain name for this host has not been set, (4) the domain name server cannot be found, (5) the domain name server does not recognize this host, (6) the command is run without super-user privileges, an error message will be sent to standard error.

If **rfstart** is used without the **–p** option and the local host has no other listing of name servers for its domain, an error message will be sent to standard error.

FILES

/usr/nserve/auth.info/*domain***/passwd**

USAGE

Administrator.

After the first use, **rfstart** will probably be used in the system startup scripts. It is expected that **rfstart** will be used with other initialization routines each time a machine is booted so that remote resources are mounted along with local resources, and are thus always available to users.

SEE ALSO

DNAME(NS_CMD), RFADMIN(NS_CMD), RFPASSWD(NS_CMD), RFSTOP(NS_CMD)

LEVEL

Level 1.

NAME

rfstop – stop Remote File Sharing

SYNOPSIS

rfstop

DESCRIPTION

The **rfstop** command stops the Remote File Sharing facility on a host until another RFSTART(NS_CMD) is executed.

Executing **rfstop** on a machine will in no way disrupt the sharing of resources among other machines engaged in Remote File Sharing. Executing **rfstop** on a machine that is not the domain name server will not halt or interrupt domain name service to other hosts in the domain. When executed on the acting domain name server, the domain name server responsibility is moved to another name server as though RFADMIN(NS_CMD) had been executed with the **–p** option. Executing **rfstop** on the domain name server does not halt domain name service to other hosts in the domain if at least one host has been previously configured as a secondary name server for the domain, and one of those hosts is currently accessible through Remote File Sharing.

This command is restricted to the super-user.

ERRORS

If (1) there are resources currently advertised by this host, (2) resources from this machine are still remotely mounted by other hosts, (3) there are still remotely mounted resources in the local file system tree, (4) **RFSTART(NS_CMD)** has not previously been executed, or (5) the command is not run with super-user privileges, an error message will be sent to standard error.

USAGE

Administrator.

SEE ALSO

ADV(NS_CMD), FUMOUNT(NS_CMD), MOUNT(AS_CMD), RFADMIN(NS_CMD), RFSTART(NS_CMD), RMNTSTAT(NS_CMD), UNADV(NS_CMD)

LEVEL

Level 1.

NAME

rmntstat - display mounted resource information

SYNOPSIS

rmntstat [–h] [*resource* **]**

DESCRIPTION

When used with no options, **rmntstat** displays a list of all local resources that are remotely mounted, the local path name, and the corresponding clients. **rmntstat** returns the remote mount data regardless of whether a resource is currently advertised; this ensures that resources that have been unadvertised but are still remotely mounted are included in the report. When a *resource* is specified, **rmntstat** displays the remote mount information only for that resource. The **–h** option causes header information to be omitted from the display.

ERRORS

If no resources are remotely mounted, **rmntstat** will return a zero exit status.

If *resource* (1) does not physically reside on the local machine or (2) is an invalid resource name, an error message will be sent to standard error.

USAGE

Administrator, End-User.

SEE ALSO

FUMOUNT(NS_CMD), MOUNT(AS_CMD)

LEVEL

Level 1.

NAME

unadv - unadvertise a resource

SYNOPSIS

unadv *resource*

DESCRIPTION

The **unadv** command unadvertises *resource*, which is the advertised symbolic name of a local directory, by removing it from the advertised information on the domain name server. Unadvertising a resource prevents subsequent remote mounts of that resource. It does not affect continued access through existing remote or local mounts.

An administrator at a server can unadvertise only those resources that physically reside on the local machine. A domain administrator, however, can unadvertise any resource in the domain by running the command from the acting domain name server and specifying the resource name as *domain.resource*.

This command is restricted to the super-user.

ERRORS

If *resource* is not found in the advertised information, an error message will be sent to standard error.

USAGE

Administrator.

If a host crashes while it has resources advertised, the domain name server may continue to list those resources as being available even though they are not available. It is only to correct this situation that a domain administrator should unadvertise another host's resources.

SEE ALSO

ADV(NS_CMD), FUMOUNT(NS_CMD), NSQUERY(NS_CMD)

LEVEL

Level 1

Indexes

General Index

D

E

F

G

N

O

P

Q

R

S

T

U

V

W

Function Index

V

W